TESIS DOCTORAL

INSTALACION DE UN SISTEMA FUNCIONAL SOBRE UN COMPUTADOR CONVENCIONAL

que se presenta en la

FACULTAD DE INFORMATICA DE MADRID

para la obtención del grado de

DOCTOR EN INFORMATICA

Autor: María Isabel García Clemente
Licenciada en Informática

Director: D. Pedro de Miguel Anasagasti
Catedrático de Computadores de la
Facultad de Informática de Madrid

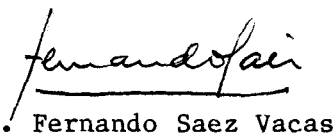
Madrid, Diciembre de 1984

TESIS DOCTORAL

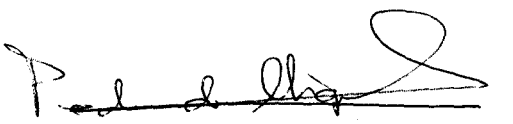
INSTALACION DE UN SISTEMA FUNCIONAL SOBRE UN COMPUTADOR CONVENCIONAL

TRIBUNAL CALIFICADOR

Presidente: D. Eugenio Andrés Puente.

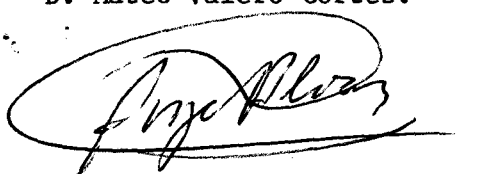


Vocales: D. Fernando Saez Vacas.



D. Pedro de Miguel Anasagasti.

D. Mateo Valero Cortés.



D. Angel Alvarez Rodriguez.

Deseo recordar a aquellas personas que con su trabajo han contribuido de forma decisíva a la realización de esta tésis.

A Pedro de Miguel, director de esta tésis, por su inestimable ayuda y sugerencias.

A Pepe Mañas, por su interés y su decisíva aportación en la parte técnica de este trabajo, y a Viri por su apoyo moral.

A Alberto Lafuente, por su contribución en la realización de los programas de simulación.

A Antonio Perez por su generosa colaboración en los detalles de presentación de este trabajo.

Al Centro de Cálculo de la Facultad de Informática de Madrid, y en particular a Juanjo Padilla por su ayuda en la puesta a punto de los programas de simulación.

A todos ellos y a los que, seguramente, olvido sin querer mi más sincero agradecimiento.

RESUMEN

En esta tesis se tratan dos aspectos fundamentales de los sistemas funcionales, de cuyo tratamiento depende, en gran medida, la eficiencia de tales sistemas. En primer lugar, se presentan y evalúan una serie de soluciones al problema de representación interna en los sistemas funcionales, soluciones basadas en la utilización de estructuras lineales para obtener una mejora en el tiempo de ejecución y en la ocupación de la memoria del sistema.

En segundo lugar, se presenta un sistema de evaluación multitarea para reducir programas funcionales, basado en la especificación de varios ficheros de salida. En él, las tareas del sistema evalúan, de forma concurrente, las diversas partes del resultado de un programa y se comunican entre sí en base a mensajes que permiten detectar y resolver, sin necesidad de abortar la ejecución, bucles de dependencias entre tareas. El sistema reacciona ante estas situaciones de error de forma comedida, de modo que este error no afecta al resto de las tareas del sistema.

Se plantea también un mecanismo de gestión de memoria, mecanismo que introduce una tarea especial en el sistema, encargada de la gestión de una memoria común a todo él, y satisface las necesidades de memoria de las tareas de forma transparente al mecanismo de evaluación.

Palabras clave: programación funcional, reducción, evaluación, representación interna, estructuras lineales, acceso directo, evaluación retardada, suspensión, recuperación frente a errores, concurrencia, programa multitarea, cálculo controlado por demanda, gestión de memoria.

ABSTRACT

Two fundamental aspects of functional systems, of which their efficiency to a large extent depends, are dealt with in this thesis.

Firstly, several solutions to the internal representation problem of functional systems are proposed and evaluated. These solutions are based on the use of linear data structures in order to achieve a shorter execution time and smaller memory requirement.

Secondly, a new multitask evaluation system for the reduction of functional programs is described; it is based on the specification of several output files. In this evaluation system, the task evaluates in the different parts of the program result concurrently. Tasks communicate one another through messages, which make it possible to detect and solve the dependency loops within tasks. The program does not need to be aborted, the sistem reacts to such error situations in a smooth way, no other tasks are affected.

A memory management system is also introduced. This management system inserts a special task, that carries-out the management of a common memory and serves all tasks memory requirements in a transparent mode to the evaluation mechanism.

Key words: functional programming, reduction, evaluation, internal representation, lineal structures, direct access, lazy evaluation, suspension, run time errors recovery, concurrency, multitask program, demand driven evaluation, memory management.

INDICE

T_i Tarea del Sistema de nombre "i".

G Gestionador de Memoria.

ST_i Estado de T_i con respecto al sistema

- EV Evaluación
- RC Recolección

SE_i Estado de T_i con respecto a la evaluación del programa

- L Latente
- A Activa
- E Espera
- M Muerto

SG_i Estado de T_i con respecto al Gestionador de Memoria

- NPG No Pendiente del Gestionador
- PPG Parada Pendiente del Gestionador

S_g Estado del Gestionador

- R Recolección
- C Concesión

S_i Pila S de la tarea T_i

D_i Pila D de la tarea T_i

PC_i Contador de programa de T_i

E_i Contexto de evaluación de T_i

FS_i Fichero de salida de T_i

BQ_i Bloque de memoria asignado a T_i

R_m Receta en evaluación de nombre m

RZ_m Receta sin evaluar de nombre m

W_m Cola de espera de la receta R_m

ME Mensajes de Evaluación

[I. T_i. R_m]
[F]

MG Mensages de Gestión

[B. T_i]
[C. b_i]
[W]
[A]

QV_i Cola de mensajes de Evaluación de T_i

QG_i Cola de mensajes de Gestión de T_i

QG_g Cola de mensajes de Gestión del Gestionador de Memoria

CAPITULO 1

INTRODUCCION

1. INTRODUCCION

Los lenguajes funcionales representan una alternatíva de muy alto nivel a los lenguajes imperatívos que dominan, en este momento, el campo de la programación de sistemas. Actualmente, las técnicas de programación funcional están adquiriendo cada vez mayor importancia debido a que el desarrollo de la tecnología permite un avance hacia lenguajes de programación de más alto nivel. Aunque existen, desde hace tiempo, lenguajes de programación cuya estructura básica es la de un lenguaje funcional, su utilización se ha restringido a aplicaciones muy concretas. Ello es debido, principalmente, a la ineficiencia de estos lenguajes, soportados por máquinas convencionales, lo que hace que su ejecución sea relatívamente lenta.

Por otra parte, la necesidad de conseguir una mayor velocidad de ejecución de los programas, ha llevado a los arquitectos de computadores a construir sistemas multiprocesador basados en el modelo clásico de Von Neumann, donde aparecen problemas importantes de sincronización entre procesos, debido al caracter temporal de los programas, cuyas instrucciones deben ejecutarse de forma secuencial.

El desarrollo de la tecnología VLSI ha hecho que el diseño de sistemas multiprocesador, se haya convertido en uno de los temas de mayor interés en la actualidad. Uno de los aspectos de mayor importancia y complejidad, relacionado con la construcción de tales sistemas, es el de cómo programarlos.

La programación de estos sistemas, utilizando lenguajes aplicatívos o funcionales constituye una de las soluciones más adecuadas, debido al caracter atemporal de tales programas y la carencia de efectos secundarios. Estas caracteristicas permiten descomponer un programa, de forma natural, en subconjuntos independientes que pueden evaluarse por separado, consiguiéndose un paralelismo transparente al programador, con problemas de sincronización más simples de resolver que en el multiprocesador clásico.

Justificación y objetívos

En esta tésis se estudian dos aspectos de los sistemas funcionales que consideramos fundamentales para la consecución del principal objetívo de nuestra linea de trabajo, que es la construcción de un sistema multiprocesador.

En primer lugar, se buscan estructuras para la representación interna de la información, que hagan más eficiente la utilización de los recursos de la máquina. Como se sabe, la representación interna de la información, en este tipo de sistemas, está basada fundamentalmente en la utilización de listas encadenadas según el modelo clásico de McCarthy [38]. Esta representación tiene sus ventajas y sus inconvenientes. En particular, en lo referente al código y al contexto de evaluación, cabe preguntarse el por qué de una representación en forma de listas encadenadas, que utiliza un mayor espacio de memoria que una estructura lineal y obliga a realizar un acceso secuencial para localizar la información.

En segundo lugar, se quiere un sistema que aproveche el paralelismo inherente a los lenguajes funcionales, basado en la utilización del concepto de "evaluación retardada" (Henderson [26]). Un sistema capaz de evaluar, de forma concurrente, diversas partes de un programa funcional y un mecanismo sencillo y potente de sincronización y comunicación entre procesos, que permita detectar y resolver las dependencias existentes entre ellos, dependencias que, caso de constituir un blucle cerrado, no supongan un bloqueo del sistema. Asimismo, se desea aportar soluciones al problema de cómo gestionar la memoria del sistema de forma eficiente.

Los resultados obtenidos constituirán una aportación importante a la hora de definir las bases del diseño y construcción del sistema multiprocesador citado.

Paralelamente, estamos desarrollando el lenguaje funcional, que pensamos podrá programar de forma eficiente este tipo de sistemas y que no se ha incluido en esta tésis por estar recogido en un informe interno [37] recientemente publicado.

Organización de este trabajo

En el capítulo 2 se hace una breve revisión de los trabajos más significativos realizados en el área de representación interna de la información, sistemas funcionales multiprocesador y recolección de restos, que han servido de base para la realización de este trabajo.

En el capítulo 3 se presenta una estructura lineal para el código, como alternativa a la representación de listas, y un juego de instrucciones completo para una máquina virtual SLO basada en la ya clásica SECD de Landin [34]. En la segunda parte de este capítulo, se introduce el concepto de "vector" para la representación del contexto de evaluación del programa, adaptándose el juego de instrucciones anteriormente expuesto al nuevo modelo de representación, dando lugar a un nuevo modelo de máquina virtual SVO.

En el capítulo 4 se trata el tema de manejo de ficheros en un sistema funcional. En él, planteamos y desarrollamos un sistema de evaluación multitarea, basado en la especificación de varios ficheros de salida. Se plantea, también, en este capítulo un mecanismo de gestión de memoria para el sistema multitarea.

El capítulo 5 se dedica a evaluar las estructuras de representación planteadas en el capítulo 3.

En el capítulo 6 se resumen las aportaciones fundamentales del presente trabajo.

Finalmente, se incluye la bibliografía referenciada y los apéndices con el juego de instrucciones planteado para el sistema de código lineal (A1) y los programas prototipo utilizados en las evaluaciones de las estructuras de representación (A2).

CAPITULO 2

ANTECEDENTES

2. ANTECEDENTES

En este capítulo se hace una breve exposición de los antecedentes en los temas que conciernen a este trabajo. Se ha dividido en dos grandes áreas. En la primera de ellas resumimos, a grandes rasgos, las ideas más significativas que intentan obtener una mayor eficiencia en la utilización de los recursos de la máquina, durante la ejecución de programas escritos en lenguajes funcionales: mejor utilización de la memoria, y reducción del tiempo de proceso. En la segunda área, trataremos el tema de la aparición y coexistencia, en un sistema funcional, de varios procesos que cooperan en la consecución del fin último: la evaluación de la expresión que es el programa funcional. Discutiremos también, en este apartado, el tema de ficheros múltiples de salida, por ser un antecedente directo de la aparición de procesos múltiples en un sistema funcional.

En esta revisión nos hemos restringido a aquellos autores cuya aportación ha sido más significativa en el avance de los distintos temas. Para un estudio mas produndo de los distintos puntos comentados, nos remitimos a las referencias citadas a lo largo de este capítulo.

2.1.- EFICIENCIA EN LA UTILIZACION DE RECURSOS DE UN SISTEMA FUNCIONAL

Son dos, fundamentalmente, los problemas a tener en cuenta en lo concerniente a este tema. El primero, reside en la ineficacia de las estructuras de listas (estructuras de datos básicas en los sistemas funcionales) en cuanto a la utilización del espacio de almacenamiento se refiere. Ello es debido a que, junto a la información propiamente dicha, es preciso almacenar apuntadores, como elementos de enlace, para conseguir el efecto de lista, lo cual conlleva el utilizar una gran cantidad de espacio adicional para representar algo que no es información, sino simplemente un modo de acceder a ella. La utilización de este tipo de estructuras tiene como consecuencia que, para acceder a un valor determinado, es preciso recorrer secuencialmente la lista en la que se encuentra, siguiendo estos apuntadores, lo cual repercute directamente en la lentitud de estos sistemas.

El segundo problema, reside en el hecho de que el procesamiento debe ser detenido periódicamente, para recuperar las celdas de memoria que ya no se utilizan y hacerlas reutilizables. Esto exige un largo proceso conocido como "recolección de restos" ("garbage collection"), que las incorpora a una lista llamada de celdas libres, lo cual incide de nuevo en el aumento del tiempo total de ejecución de un programa. Trataremos a continuación, y por separado, ambos temas, así como las soluciones aportadas por diversos autores.

2.1.1.- Técnicas de representación de listas

Desde que McCarthy [38] introdujo la notación de listas para representar las expresiones S en el lenguaje LISP, primer lenguaje inspirado en el cálculo lambda y punto de partida de los lenguajes funcionales, estas estructuras se han venido utilizando como forma básica de representar la información en este tipo de sistemas. Una lista es un átomo o bien, un par ordenado de objetos que son a su vez listas.

Entre los motivos por los que se utilizaron este tipo de estructuras destacamos los siguientes: 1) el hecho de que tanto el tamaño como el número de expresiones que maneja un programa son en principio impredecibles, 2) la existencia de información que deja de ser util en un momento determinado, pudiendo de esta forma ser reutilizadas las celdas de memoria que la contienen si se añaden a una lista de celdas libres, y por último, 3) el hecho de que una expresión puede ser compartida por varias expresiones. Sería, por tanto, muy dificil utilizar bloques de longitud fija para satisfacer este tipo de necesidades.

Las listas tienen, sin embargo, una desventaja fundamental, que es, como dijimos antes, su ineficaz utilización del espacio de almacenamiento que las soporta, tanto en volumen como en el lento y complejo acceso a sus componentes.

Son muchos los trabajos realizados con el fin de paliar este inconveniente. En algunos casos se dan soluciones parciales, mientras que en otros, se trata de buscar representaciones alternativas. Hacemos a continuación una revisión de los que consideramos más significativos.

Como es sabido, la representación interna de las listas es una representación etiquetada, es decir, cada uno de sus componentes lleva asociada una información acerca de su tipo (Henderson [26]), siendo los tipos básicos el tipo átomo y el tipo constructor de listas "cons". Por ejemplo, un elemento, que representa un constructor de listas, tiene una parte de tipo que indica que es un objeto de tipo "cons" y otra parte constituida por un par de apuntadores a dos objetos, que son el primer elemento de esa lista o "car" y el resto de ella o "cdr".

Entre las soluciones dadas al problema que nos ocupa, aparece la notación de "punteros con tipo", introducida en algunos prototipos de máquinas Lisp (Steele [42], Sussman [43]), de forma que a cada puntero se le asocia una información de tipo, que identifica la naturaleza de la información a la que apunta. En un apuntador, cuya parte de tipo indique "número entero", se utiliza la parte de dirección como dato inmediato, es decir, la parte de dirección contiene el valor mismo del número. Esto puede hacerse extensible a otros tipos de objetos atómicos, consiguiéndose, de esta forma, una serie de ventajas entre las que se pueden citar:

- Los objetos de tipo "entero" ocupan una celda de memoria menos, con lo cual se ahorra tanto en espacio como en tiempo de acceso.

- Las operaciones primitivas aritméticas y lógicas disminuyen su tiempo de ejecución, al disminuir el número de accesos para acceder a los operandos, y al ser mas rápida la comprobación de compatibilidad de los tipos de sus operandos.

- Las operaciones primitivas de comprobación de tipos: átomo, número,... disminuyen su tiempo de ejecución, ya que los tipos van junto a los operandos.

La mejora obtenida con esta representación es muy significativa, tanto en lo referente a espacio como a tiempo.

Más significativos son, sin embargo, los resultados obtenidos utilizando técnicas de codificación compactadas y linealización de listas, orientadas a la obtención de un menor gasto de memoria, existiendo gran

cantidad de estudios comparatívos en este campo. En lo que sigue hemos utilizado básicamente los trabajos realizados por Hansen [24], Clark [8] y Bobrow [5], que creemos cubren ampliamente estos temas.

Ya que, en principio, la mayor parte de las implantaciones de las estructuras de listas utilizan campos de longitud fija, para contener tanto datos como apuntadores, Hansen [24] realiza un primer intento de reducir el tamaño de estos campos, codificándolos de forma que los valores más probables ocupen menos espacio que los menos probables. Según estudios realizados sobre la utilización de la estructura de listas en LISP (Clark [8]), se ha comprobado que algunos de los valores de estos campos, aparecen con mayor probabilidad que otros, por ejemplo, en el caso de los punteros a listas, los resultados obtenidos muestran, que la mayoría de las veces, la celda que contiene el puntero está muy próxima físicamente en memoria a la celda a la que apunta. Esto se debe al hecho de que estas celdas se crean en instantes de tiempo muy próximos.

Esta regularidad sugiere una manera simple y natural de codificar los apuntadores, como es la de utilizar un campo pequeño para indicar un incremento sobre la dirección de la celda que contiene el apuntador. Ya que este incremento es la mayoría de las veces muy pequeño, bastará con un número reducido de bits para representarlo. En los casos en que los punteros necesiten un campo de mayor longitud, se puede siempre recurrir a una combinación de bits que indique que la dirección real está en otro lugar de memoria, por ejemplo, en una tabla hash cuya clave sea la dirección de la celda que contiene al apuntador.

No es tan sencilla, sin embargo, la codificación de punteros a átomos, ya que su frecuencia de aparición depende en gran medida del programa de que se trate, siendo realmente difícil asegurar, antes de la ejecución del programa, qué átomos serán los más referenciados posteriormente.

A consecuencia de la tendencia de los punteros a listas de apuntar a celdas próximas físicamente, y dado que esta distancia es muy frecuentemente de sólo una unidad, surge la utilización de la técnica de linealización de listas. En lo que sigue, hablaremos de la linealización por el

"cdr" o "cdr-coding", siendo análogos los planteamientos para la linealización por el car ("car-coding").

La linealización de una lista, es el proceso de reorganizarla de forma que sus "cdrs" apunten a la siguiente posición secuencial, siempre que esto sea posible. Este proceso no siempre puede realizarse, ya que si los cdr de dos listas diferentes apuntan a la misma celda, solo puede ser adyacente a una de ellas.

La representación compactada, que vimos anteriormente, gana en eficiencia si se utiliza la linealización. Por ejemplo, en un sistema con "cdr-coding" bastarían dos bits para indicar:

- NIL
- siguiente posición de memoria
- el valor real está en otro lugar

Este es el método utilizado por Van der Poel [47].

Las medidas realizadas por Clark [9] sobre un conjunto de programas prototipo muestran, que en el 98% de los casos, el cdr de una lista apunta a la siguiente posición de memoria. Ya que los punteros a listas son mas frecuentes en el "cdr" que en el "car", parece ser éste el método que proporciona mayor ahorro.

El proceso de linealización se asocia normalmente al de recolección de restos, aunque estas estructuras podrían ser obtenidas tambien durante la ejecución normal. Para ello se necesitaría diseñar apropiadamente las primitivas del sistema.

Este tipo de representación ha demostrado ser de gran utilidad en sistemas con memoria virtual. Al verse incrementado el número de apuntadores a celdas dentro de una misma página y aumentar el número de celdas que caben en ella, se reduce el número de referencias fuera de página y el tiempo consumido por fallos de página, de esta forma se aumenta la velocidad de proceso. Como contrapartida a estas ventajas, proporcionadas por la linealización y compactación de listas, está el inconveniente del tiempo

consumido en el proceso de linealización (Clark [8]).

Aunque los métodos de linealización aprovechan la contiguidad para obtener una mayor eficiencia en cuanto a utilización de memoria, nada hacen por mejorar el tiempo de acceso en aplicaciones tales cómo: acceder a un elemento por su posición en una lista (índice), acceder con la misma facilidad a los elementos de uno y otro extremo de la lista, o realizar una concatenación rápida de listas. Keller [31] plantea, como alternatíva, el esquema de representación de listas "CONC", que no sustituye totalmente al tradicional "CONS", coexistiendo con él. Con ello se pretende una mejor utilización de la contiguidad, tanto en cuestión de espacio como de tiempo.

Mientras que "cons" crea pares de apuntadores (car y cdr), "conc" crea tuplas de tamaño arbitrario. La introducción del operador "conc", viene acompañada por la aparición de una nueva función primitíva "tlenght", que da la longitud de la tupla, la cual se almacena dentro de ésta, y de una función "tselect" de dos argumentos: un índice entero y una tupla, de forma que:

si X es una tupla $(X_1, X_2, \ldots, X_n)$
entonces

$$\text{tlenght}(X) = n$$

y el índice para "tselect" debe pertenecer al conjunto de valores

$$\{ 1, 2, \ldots, n, -1, \ldots -n \}$$

$$\text{tselect}(X, i) = \begin{cases} X_i & \text{si } i > 0 \\ X_{n+i+1} & \text{si } i < 0 \end{cases}$$

Para relacionar estos conceptos con los tradicionales "car" y "cdr", se puede definir:

$$\text{car}(X) = \text{tselect}(X, 1) = \text{tselect}(X, -n)$$
$$\text{cdr}(X) = \text{tselect}(X, n) = \text{tselect}(X, -1)$$

que es consistente con la definición tradicional cuando n= 2. Esta función permite, como se puede comprobar, acceder con la misma facilidad, al primer elemento de la tupla, tselect (X, 1), y al último, tselect (X, -1).

Una ventaja inmediata de esta representación, es la introducción del concepto vector en un sistema funcional. Este concepto aprovecha la característica de las memorias de poder acceder con la misma facilidad a cualquiera de sus posiciones (no se incluye la posibilidad de poder almacenar de forma destructíva en el vector). Esto elimina la necesidad de añadir una nueva clase de objetos "array" como se ha hecho en algunos dialectos Lisp. Las listas de tamaño fijo pueden representarse como tuplas y ser ubicadas, por tanto, en posiciones contiguas de memoria, consiguiéndose así directamente una mejor utilización de la memoria, ventaja perseguida igualmente por otras técnicas como el "cdr-coding".

Al concatenar dos listas, el hecho de conservar la representación contigua para el resultado llevaría consigo el tener que copiar todos sus elementos en el espacio contiguo. Keller prorpone una concatenación virtual utilizando nuevos tipos de objetos:

- tupla: lista de objetos almacenados en posiciones contiguas de memoria.

```
tupla:= record
   longitud: entero;      (* número de elementos *)
   dirección: entero;     (* dirección primer elemento *)
        end
```

- conc: concatenación de dos subobjetos.

```
conc:= record
   longitud: entero;      (* longitud total de la lista resultante *)
   tupla: tupla;          (* sus componentes apuntan a los elementos
                             o listas concatenados *)
      end
```

y nuevas primitivas, que no citaremos aquí, a las que se hace referencia en (Keller [31]).

2.1.2.- RECOLECCION DE RESTOS

Uno de los mayores problemas existentes en los sistemas funcionales, y en general en los sistemas de procesamiento de listas, es el proceso conocido como "recolección de restos" (garbage collection). Surge a consecuencia del gran consumo de memoria que tienen estos sistemas, y debido a que el espacio de direcciones de que se dispone es menor que el necesario para la ejecución de los programas, resultando imprescindible la inclusión de un mecanismo que identifique las celdas de memoria que dejan de ser útiles y haga posible su reutilización.

Aunque son muchos los algorítmos de recolección existentes, las técnicas de recolección son básicamente de 3 tipos:

explícita

Consiste en dejar al programador la labor de decidir en qué momento debe hacerse la recolección. Esta fué la técnica utilizada en los primeros sistemas de procesamiento de listas (Newel [40]). Ha sido desechada en la actualidad, tanto por la carga que supone para el programador como por su excesiva propensión al error.

cuenta de referencias

Consiste en utilizar un contador de referencias (reference-count) para cada celda, que indique el número de celdas que le apuntan, (Weizenbaum [49]). Cuando éste llega a cero, la celda queda automáticamente liberada para su posterior utilización. Esta técnica, tiene una serie de inconvenientes, entre los que se pueden citar, el espacio adicional necesario para almacenar el contador (que para celdas de pequeño tamaño puede llegar a ser de hasta un 25% más (Steele [41])) la necesidad de, que las primitívas del sistema deban actualizar contínuamente estos contadores, y por último, la dificultad de averiguar cuando una lista que hace referencia a sí misma (lista circular) es ya innecesaria.

recolección

La tercera técnica, la más comunmente utilizada, es la propuesta por McCarthy [38] y utilizada en el sistema LISP 1.5 [39]. Consiste

en ignorar el problema hasta que se acaba la lista de celdas disponibles (freelist). Cuando esto ocurre, el proceso se detiene temporalmente, mientras la rutina de recolección identifica qué celdas no son ya útiles (celdas no accesibles), y las incorpora a la lista de disponibles. Esta técnica garantiza la recolección de listas circulares sin dificultad y sólo necesita añadir uno o dos bits a cada celda, liberando al programador y a las primitívas del sistema de esta tarea. Sólo las primitívas que construyen nuevas estructuras, a partir de celdas disponibles, pueden necesitar llamar al recolector.

Debido a sus ventajas, ha sido esta última técnica, también llamada de "marcado y barrido" (mark and sweep), la más estudiada en los últimos tiempos y la utilizada, con algunas variaciones, en la mayor parte de los actuales sistemas de procesamiento de listas.

En cuanto a los algorítmos de recolección, su esquema básico se compone de los siguientes pasos:

I.- [MARCAR]
Marcar las celdas accesibles.

II.- [COMPACTAR]
Compactar las celdas marcadas en un espacio contiguo de memoria (si como suele suceder se utiliza alguna técnica de compactación) y actualizar los apuntadores a ellas.

III.- [CONSTRUIR]
Formar con las celdas no accesibles el nuevo espacio de memoria disponible.

El primer paso se lleva a cabo, generalmente, mediante un simple método recursívo que identifica las celdas accesibles a partir de la información contenida en los registros de la máquina (raices de las listas utilizadas), marcandolas seguidamente.

En la realización del segundo paso, los métodos más utilizados son los métodos de "copiado" o de utilización de "dos apuntadores". El método de copiado consiste en utilizar dos areas de memoria. Durante la ejecución normal del programa se trabaja en una de ellas ("fromspace") y al comenzar

la recolección, las celdas accesibles se mueven a la otra ("topspace"). Al finalizar la recolección, sólo quedaran en la primera área las celdas no accesibles y continuará la ejecución normal sobre la segunda (Baker [3]). Este método ha sido estudiado principalmente para su aplicación en sistemas con memoria virtual (Fenichel [15]).

En particular, en el algorítmo de Baker [3] las dos primeras fases se realizan a la vez. Las celdas accesibles se marcan y se mueven al "topspace", guardándose en la antigua dirección la dirección a la que se ha movido su información. Siempre que se encuentre un apuntador a una celda ya movida, se actualiza éste.

En el método de dos apuntadores (Hart [25]), es preciso realizar dos recorridos por la memoria. En el primero se utilizan dos apuntadores, uno al final de la memoria y otro al principio. Este último se va incrementando, hasta encontrar una celda no marcada. En este punto, el puntero al final de la memoria se decrementa hasta apuntar a una celda ya marcada. y se mueve el contenido de la marcada a la no marcada, colocándose en la celda movida la nueva dirección. Cuando ambos apuntadores se encuentran, todas las celdas se han compactado en la parte superior de la memoria. El segundo recorrido se realiza para reajustar los apuntadores a las celdas. Ya que algunas celdas se han movido de lugar, es necesario actualizar los apuntadores a las celdas que han quedado obsoletas. Este recorrido se realiza sólo sobre el área compactada.

El tercer paso del algorítmo, puede ser realizado de dos formas diferentes (Cohen [12]); incorporando las celdas no accesibles a una lista de celdas libres, en la que estas celdas están enlazadas por apuntadores, o bien, compactando todas las celdas accesibles en un extremo de la memoria, de modo que en el otro extremo estará la zona libre formada por palabras consecutivas.

La principal desventaja de los recolectores clásicos, es el hecho de que el proceso normal debe ser detenido impredeciblemente, durante un tiempo proporcional al número de celdas accesibles en un momento dado. En aplicaciones para tiempo real, el problema más importante es el de la impredecibilidad del tiempo de recolección (Dijkstra [14]).

Ya que la recolección debe hacerse, la única forma de evitar la suspensión del programa es introducir paralelismo, es decir, efectuar la recolección mientras se realizan las operaciones normales sobre listas. Knuth [32] atribuye esta idea a Minsky. El método más sencillo, sería compartir el tiempo de procesador entre el proceso normal y la recolección, bien en base a interrupciones de entrada/salida o en base a intervalos de tiempo fijados por un reloj. Otro método, más adecuado, es el de utilizar dos procesadores, uno para el proceso normal (también llamado proceso "mutador") y otro para el proceso de recolección (proceso "recolector"). De esta forma, ambos pueden ser realmente simultáneos (Steele [41]).

El paralelismo introduce nuevos problemas, ya que los procesos están accediendo a las mismas listas para realizar labores diferentes sobre ellas. Por ejemplo, el mutador coge celdas de la lista de libres para formar nuevas estructuras, mientras que el recolector añade nuevas celdas a esta lista. Es entonces imprescincible incluir mecanismos de sincronización y comunicación entre ambos procesos. De entre los problemas más evidentes que plantea este paralelismo de tareas podemos citar:

- Que el proceso mutador cree una nueva estructura a partir de celdas libres. La fase de marcado debería saber por tanto, que ahora estas celdas son accesibles, ya que de no ser así, el recolector las consideraría finalmente como "restos".

- Que se altere un apuntador de una celda ya marcada para pasar a apuntar a una no marcada. Como en el caso anterior, podría ocurrir que esta última permaneciese sin marcar, siendo reincorporada erróneamente por el recolector a la lista de libres.

En cualquier caso, el proceso mutador debe conocer en qué fase se encuentra el recolector y actuar en consecuencia, siendo por tanto necesaria una estrecha colaboración entre ambos.

La solución aportada por Steele [41] consiste en utilizar dos bis que identifiquen el estado de recolección en que se encuentra cada celda. Con ellos es posible conocer si una celda es accesible, si ha sido trasladada a otro lugar, o si ha sido desechada, o está ya en la lista de li-

bres. Lo más complejo de este método, en realidad, es la actualización correcta de estos dos bits.

Dijsktra [14] describe el marcado de celdas en base a colores. Inicialmente todas las celdas son de color blanco; la acción combinada del mutador y la fase de marcado del recolector, hace que todas las celdas accesibles sean de color negro. Al finalizar esta fase, todas las que permanezcan blancas serán consideradas como reutilizables. Durante la fase de marcado se debe cumplir, que ninguna celda pase de tener color negro a tenerlo blanco. Una relación invariante encontrada para este proceso, que es incialmente cierta y se cumple a lo largo de todo el proceso, es la siguiente:

P1: no existe ningún apuntador de una celda negra a una blanca.

Es pues necesario evitar que el mutador introduzca un puntero de una celda negra a una blanca, ya que violaría esta invariante. El mutador no puede pasarla a negra directamente, ya que violaría de nuevo P1 entre esta celda y sus sucesoras inmediatas. Se introduce entonces un nuevo color, el gris, permitiéndose que el mutador cambie blanco por gris. Las celdas grises deben convertirse finalmente en negras, pero tienen aún sucesoras blancas, así pues, la fase de marcado debe hacer negras las celdas grises y cambiar el color de sus sucesoras hasta que no haya celdas grises. Evidentemente, para que no se produzcan errores, la operación de cambiar el color de una celda debe ser una operación indivisible.

Lamport [33], plantea una variante del método anterior para el caso en que existan múltiples mutadores, la solución se basa en la paralelización de las dos fases básicas de la recolección: marcado y barrido. La fase de marcado se realiza mediante la cooperación de varios procesos concurrentes, cada uno ocupado de marcar un conjunto determinado de celdas, siendo necesario un mecanismo de sincronización entre ellos. Lo mismo hace con la fase de barrido, donde múltiples procesos, sincronizados entre sí, realizan la labor de determinar qué celdas no son ya accesibles y pueden ser por tanto reutilizadas.

Posteriormente, incluye el paralelismo entre ambas fases, mediante

una ejecución "pipeline" de ambas, es decir, realizando la (i+1)-ésima ejecución de una fase de marcado concurrentemente con la i-ésima ejecución de una fase de barrido. Surge aquí un nuevo problema, debido a que todas las celdas deben ser de color blanco antes de comenzar la fase de marcado. Pero, el proceso que realiza el barrido, espera que las celdas de color blanco sean restos. La solución aportada por Lamport [33], consiste en introducir un nuevo color, el púrpura, de forma que, antes de ejecutar ambas fases, las celdas de color blanco pasan a ser púrpura y las de color negro a blanco. Los procesos encargados de realizar el barrido de la memoria, recolectan las celdas de color púrpura, mientras que los procesos que realizan el marcado ignoran estas celdas. Los mutadores no pueden nunca hacer que una celda apunte a otra de color púrpura, por lo que es fácil comprobar que una celda gris nunca apuntará a una púrpura. El algoritmo consta pues de cuatro pasos que se repiten cíclicamente:

1. Esperar a que los procesos marcadores y recolectores paren.
2. Cambiar el color de las celdas de blanco a púrpura y de negro a blanco.
3. Compartir todas las raices de las estructuras utilizadas por el programa, esto es, cambiar su color a gris.
4. Arrancar los procesos marcadores y recolectores.

Para que el algorítmo sea eficiente, los pasos 2 y 3 deben ser rápidos, ya que, durante este tiempo, los procesos que realizan el marcado y el barrido de la memoria permanecen parados.

Estos algorítmos llevan consigo diversos criterios de exactitud y relaciones invariantes que se estudian con detalle en las referencias utilizadas (Dijkstra [14], Lamport [33], Steele [41]).

2.2.- MULTIPROCESO

Muchos son los autores que han estudiado el tema de la utilización de múltiprocesadores para la ejecución de una tarea, posibilidad especialmente interesante por el desarrollo de las técnicas VLSI, que proporcionan los medios adecuados para obtener un gran número de elementos idénticos a bajo costo. Uno de los temas de mayor importancia y complejidad relaciona-

do con este hecho es la programación de tales sistemas. Cómo repartir el trabajo eficientemente entre los procesadores y cómo detectar el paralelismo existente entre las diversas partes del programa, son las preguntas a responder.

Friedman y Wise [19] señalan que los lenguajes funcionales o aplicatívos son especialmente adecuados para la programación de tales sistemas, debido, tanto a su estructura, como a su carencia de efectos secundarios ("side-effects"). Los teoremas de Church-Rosser prueban que el valor de una expresión aplicatíva no varía a pesar del orden o de la velocidad relativa de evaluación.

2.2.1.- OBTENCION DE PARALELISMO

Una forma de obtener paralelismo, en un programa funcional, es la evaluación simultánea de los argumentos formales de la función, pudiéndose asignar un proceso a cada uno de estos argumentos (procesos hijos) y otro a la función (proceso padre). Surge de esta idea el concepto de evaluación "impaciente" ("eager evaluation") estudiado por varios autores [2] [22] [23], en la que se arranca la evaluación de subexpresiones tan pronto como es posible, tanto si su valor va a ser utilizado posteriormente como si no.

El problema que se plantea, en este tipo de sistema, es que pueda asignarse un proceso a una expresión cuyo valor no sea utilizado para la obtención del resultado final. A este tipo de procesos se les llama "irrelevantes". Si no hubiese forma de determinar qué procesos son irrelevantes en el sistema, este tipo de procesos supondrían un desperdicio significativo de la potencia de cálculo del sistema, y si se tratase de evaluaciones que no acabasen nunca, el problema sería aún mayor ya que esta potencia se perdería para siempre.

Dos son las soluciones planteadas para este problema. La primera, (Baker [2]) es la de ampliar el proceso de recolección de restos para que, del mismo modo que se identifican las celdas de memoria que ya no son útiles, se identifiquen los procesos que son irrelevantes en el sistema. La clave de la recolección de procesos es la existencia de un conjunto de pa-

labras dentro de la memoria común a todos los procesadores, donde quede reflejado el estado de cada uno de los procesos. Marcar un proceso equivale a decir que es relevante para el sistema, y puede reemprender por tanto su ejecución tras la recolección. Un proceso no marcado es recuperado por el recolector y, tras la recolección, no se le asignará procesador alguno, desapareciendo del sistema.

Grit y Page [22] [23], ven el sistema como un árbol de tareas en el que cada nodo representa la llamada a una función y sus descendientes representan la evaluación de sus argumentos. El mecanismo sugerido por estos autores, para eliminar del sistema las tareas irrelevantes, es la inclusión de un tipo de tareas especiales llamadas "asesinas" ("killer tasks"). Cuando se encuentra un subárbol de tareas irrelevante se arranca automáticamente una de estas tareas para eliminar su raiz. Esta, a su vez, arranca nuevas tareas "asesinas" para cada subárbol de esta raiz, hasta acabar con todos los procesos de esa estructura.

Este tipo de evaluación se dice que, está "conducida por los datos" ("data-driven"), en el sentido de que las tareas llevan a cabo su labor a una velocidad determinada por la disponibilidad de los argumentos que necesita. De esta forma, una función puede tratar de acceder a un argumento y no estar aún evaluado. En este caso, el proceso que evalúa la función se pondrá en una cola de espera y será "despertado" cuando termine la evaluación del argumento en cuestión. Baker realiza esta idea, creando para cada argumento lo que denomina "future", caracterizado por 3 tipos de información:

- el proceso que evaluará el argumento, que corresponde a la identificación del procesador virtual que realizará la evaluación.

- una dirección de memoria, donde este proceso colocará el resultado de la evaluación.

- una cola de procesos (FIFO), que contenga los procesos que esperan ese valor.

Friedman y Wise [16] [21], estudian la inclusión, en el lenguaje,

de estructuras que potencien la ejecución paralela, de forma que sea el compilador quién detecte la posibilidad de paralelismo en un programa. Estos autores utilizan el concepto de "suspending cons", que permite un paralelismo masívo, no estructurado, en un sistema con miles de procesadores, y el de "combinación funcional" que permite expresar de forma natural funciones que devuelvan más de un resultado.

La primitiva CONS, en lugar de evaluar sus argumentos, y crear la estructura completa, no los evalua, crea una estructura que consta de dos "suspensiones". Una suspensión consiste en una referencia al código necesario para evaluar el argumento y una referencia al contexto (valor de las variables) en el que se crea. Estas dos informaciones permanecen invariables durante la vida de la suspensión. El acceso a esta estructura, por parte de las primitivas "primero" (car) y "resto" (cdr), hace que se arranque su evaluación, sustituyéndose en último término la suspensión por su valor (valor obtenido de evaluar una parte de código en un contexto dado). Cualquier acceso posterior a esta celda encontrará dicho valor.

Un sistema con este tipo de primitiva puede tener, durante el curso de la evaluación, cientos de suspensiones pendientes. Si no existiese más que un procesador, todas ellas deberían esperar a que una primitíva del tipo "car" o "cdr" arrancase su evaluación. En un sistema multiprocesador, estas suspensiones pueden evaluarse simultáneamente, sin retrasar la evaluación principal. Es en este punto donde aparecen los procesos llamados "coroneles" y "sargentos", dependiendo de la importancia de la tarea que llevan a cabo.

El proceso "coronel" se comporta como en el caso monoprocesador, excepto que de vez en cuando accede a lo que debería haber sido una "suspensión" y en lugar de ello encuentra el resultado que le ha proporcionado un proceso "sargento" que "pasó" antes por allí. Los procesos "sargentos" siguen al proceso "coronel", satisfaciendo por anticipado sus necesidades. Surge de nuevo el hecho de que el esfuerzo de alguno de ellos sea vano, ya que su labor no sea nunca necesaria. Si resulta además que es un proceso que no acaba, se perderá en el sistema hasta que alguien detecte que es un proceso irrelevante, y por consiguiente, lo elimine del sistem liberando al procesador asociado a él. Friedman y Wise sugieren que esta labor sea

realizada, por el recolector, de la misma forma que lo hacía Baker.

2.2.2.- COMBINACION FUNCIONAL: FICHEROS MULTIPLES DE SALIDA.

La combinación funcional es, como dijimos en el apartado anterior, una herramienta introducida en el lenguaje para representar funciones que devuel un resultado. Nos hemos basado, para esta exposición, en los trabajos realizados por Friedman y Wise [17], [19], [20] [21].

Existen tres soluciones clásicas al problema de especificar funciones recursívas que devuelvan más de un resultado. La primera, no permitida en programación aplicatíva, consiste en utilizar variables globales o parámetros pasados por referencia. La segunda alternatíva, menos evidente, consiste en la utilización de funciones que se pasan, como argumentos, a funciones secundarias que las aplican a otros parámetros. Aunque en los lenguajes funcionales está permitido el uso de funciones de segundo orden, esta solución no parece apropiada, ya que se busca algo más transparente y sencillo para el usuario. La tercera solución clásica, más cercana al concepto que nos ocupa, utiliza una lista o un registro que contenga los diversos resultados. Cada llamada recursíva a la función, debe descomponer, modificar y volver a componer dicha estructura.

La solución aportada por estos autores es semánticamente equivalente a esta última alternatíva, pero no necesita que el programador especifique la descomposición y recomposición de la estructura resultante.

La llamada a función se representa, según la notación utilizada por estos autores, como:

<f, 1>

donde el primer elemento "f" representa la operación (función) que se quiere aplicar y "1" es el argumento al que se aplica, que en general será una lista.

La combinación funcional consiste en ampliar la estructura anterior a la más general:

$$< [f_1\ f_2\ \ldots\ f_m],\ [l_1\ l_2\ \ldots\ l_n] >$$

La marca de una combinación funcional es, por tanto, la aparición de una lista en la posición asignada a la función. A esta lista se le llama "combinador" y se supone que cada f_j es una función válida, pudiendo ser a su vez un combinador. Cualquier f_j necesita como máximo n argumentos, cada uno de los cuales procede de la estructura de los argumentos del combinador. Los argumentos l_i se pueden ver como las filas de una matriz que se pasa al combinador por columnas. La longitud de la lista resultante es el mínimo de la longitud del combinador y de todos sus argumentos. Veamos esto con un ejemplo sencillo. Sea, por ejemplo, una combinación funcional que obtiene la suma y el producto de los elementos de una serie de listas:

```
< [+ *] [[1 2] [4 5] [8 #]] >
```

f_1= +, f_2= *
l_1= [1 2]
l_2= [4 5]
l_3= [8 #]

Podemos representar esta expresión, para ver mejor su efecto, cómo:

```
< [+ *] [
  [1 2]
  [4 5]
  [8 #] ] >
```

cuyo resultado será la lista de dos elementos (13 7), y dónde el símbolo "#" representa un parámetro, que en este caso no existe y que actúa como elemento neutro de la operación correspondiente.

La combinación funcional proporciona un tipo más de paralelismo, evidentemente limitado, como resultado de arrancar un proceso para obtener cada uno de los resultados; en el caso de nuestro ejemplo uno para cada una de las operaciones primitivas "+" y "*".

Múltiples ficheros de salida

Como ya es sabido, los lenguajes puramente funcionales no poseen funciones que indiquen explícitamente lectura o escritura, las clásicas "read" y "write" de los lenguajes convencionales. Esto es debido a que estas funciones incluyen efectos secundarios sobre el estado de los ficheros y dispositívos asociados, con implicaciones en futuras llamadas a dichas funciones (esto viola la propiedad de "transparencia referencial" característica de los lenguajes aplicatívos).

Por otra parte, la utilización de ficheros, tanto de entrada como de salida, es evidentemente necesaria, ya que el usuario quiere generalmente que su programa trabaje sobre diferentes datos, sin tener que recompilar cada vez el programa, y además desea tener constancia de los resultados de dicho programa, bien sea en un fichero en disco o en la propia pantalla.

Friedman [19] propone que los ficheros sean los argumentos y resultados de la función principal del programa. De este modo, la especificación de ficheros de entrada queda restringida a la cabecera del programa. Con el concepto de "combinación funcional" se consigue que el resultado del programa sea un fichero de ficheros, cuya especificación es externa al lenguaje, de forma que en cada uno de ellos quede reflejado uno de los resultados del programa.

Estos autores hablan de "promesas de fichero", ya que incluyen en estas estructuras el concepto de "suspensión", al que hicimos referencia anteriormente, en este caso las estructuras suspendidas son los mismos ficheros. Los ficheros de salida son, en principio, "promesas" (expresiones no evaluadas). A medida que el driver va necesitando conocer la información que contiene para escribirla, va aplicando las primitívas "primero" y "resto", provocando que se realicen las evaluaciones oportunas. De este modo, el proceso de evaluación de expresiones se solapa con la salida, ya que sólo se evalúa lo necesario para obtener la información que se va a escribir. A este tipo de evaluación se le conoce como "evaluación dirigida por la salida" ("output driven").

Con relación a los diferentes drivers de los ficheros de salida, cada uno podría evolucionar con independencia de los demás, a pesar de acceder a estructuras comunes. Parece más adecuado sincronizarlos de algún modo, ya que si uno de ellos pone de manifiesto totalmente su fichero y otro lo deja "suspendido" (por ejemplo, el correspondiente a un dispositivo de acceso aleatorio), puede darse el caso de que contextos que dejan de ser últiles debido a las evaluaciones correspondientes al primer fichero pueden ser referenciados por suspensiones del segundo fichero, debiendo por tanto conservarse. Una solución es realizar las evaluaciones correspondientes a ambos ficheros concurrentemente para que tal información no deba ser conservada necesariamente.

Existe el problema de que la evaluación del segundo fichero no sea convergente, a pesar que la del primero si lo sea, o que la evaluación del segundo necesite contextos distintos a los que se necesitan para evaluar el primero. La sincronización podría hacerse adecuando la evaluación de ambos ficheros al más rápido de los drivers. Friedman [19] propone una solución que consiste en evaluar las suspensiones antes de que éstas sean necesarias, es decir, cuando un contexto es abandonado por la evaluación de una suspensión, hacer que todas las suspensiones que hagan referencia a él sean evaluadas ("dragging"). Esto haría que la evaluación de parte de un fichero llevase consigo la evaluación de parte de otros ficheros.

CAPITULO 3

PROBLEMAS DE REPRESENTACION INTERNA

3. PROBLEMAS DE REPRESENTACION INTERNA

Presentamos en este capítulo nuestras soluciones a uno de los problemas básicos de los sistemas funcionales, como es el de la representación de la información dentro de la máquina, debido a la gran repercusión que ésta tiene sobre el rendimiento del sistema, tanto en cuestión de utilización del espacio disponible como en la velocidad de ejecución.

Partimos de un modelo de máquina virtual basada en la ya clásica SECD de Landín [34] utilizada por diversos autores de gran relevancia por sus aportaciones al tema de lenguajes y arquitecturas funcionales (Burge [6], Turner [45], Henderson [27]). Su nombre procede de sus cuatro componentes básicos: una pila S, un contexto E, un código C y una pila D, y la única forma de representación interna es la clásica representación de listas introducida por McCarthy [38]. Una vez hecha una revisión de las ineficiencias que en general proporciona este tipo de estructura, y que hemos resumido en el capítulo anterior, proponemos dos soluciones, en cuanto a representación interna se refiere.

La primera, se refiere a la representación del código a ejecutar. Proponemos una representación lineal, similar a la utilizada en las arquitecturas clásicas, es decir, almacenando las instrucciones de máquina en palabras consecutivas de memoria, y un juego de instrucciones basado en Henderson [27] y Burge [6], ciertamente alejado de juegos de instrucciones tan complejos como los utilizados por Keller [30] o en la máquina Lisp del MIT [48]. Con ello hemos obtenido una mejora significativa, tanto en tiempo de acceso como en ocupación de memoria, según hemos podido comprobar por las evaluaciones realizadas y cuyos resultados se muestran en el capítulo 5. Hay que hacer notar que lo que se pretende con esta representación es un flujo de control en la máquina y no en el lenguaje, que es precisamente lo que se critica de los lenguajes convencionales y por lo que éstos son ineficientes en cuanto a capacidad expresiva se refiere.

La segunda solución se refiere a la representación del contexto, o correspondencia nombres-valores. Esta representación se viene realizando

mediante una lista que contiene los valores definidos en cada momento (Henderson [27]). Planteamos aquí una representación, apoyándonos en el concepto de "vector", que permita un acceso rápido a los diversos valores de dicho contexto por parte de las instrucciones de máquina.

Trataremos en este capítulo ambos temas por separado. Dedicaremos un primer apartado a definir algunos de los conceptos utilizados a lo largo de la exposición, con el objeto de fijar algunas ideas que creemos de utilidad para su mejor comprensión. En el segundo apartado trataremos la cuestión de representación del código, así como el plantamiento de un juego de instrucciones para la máquina virtual. Finalmente, pasaremos, en el tercer apartado, a tratar todo lo referente a la representación del contexto, estudiando las modificaciones que esta representación introduce en el juego de instrucciones planteado con anterioridad.

3.1.- CONCEPTOS

3.1.1.- COMPONENTES BASICOS

Los cuatro componentes básicos de la máquina virtual que sirve como soporte al desarrollo realizado y a los que hicimos referencia anteriormente son los siguientes:

S : pila de datos intermedios y resultados
E : contexto accesible en cada momento
PC : contador de programa
D : pila de labores pendientes o pila de estado

- S y D, en nuestro modelo son dos pilas que comparten un espacio común de memoria y van creciendo una hacia la otra. S se utiliza para guardar resultados intermedios durante la ejecución del programa y D para guardar información sobre el estado de la máquina en el momento de la llamada a una función. Esta zona de memoria tiene una longitud de palabra capaz de almacenar tanto la dirección en la memoria de celdas de la información que guardan como el tipo de esta información, siguiendo de esta forma la notación de "punteros con tipo".

Utilizaremos para las pilas la notación siguiente:

x,s representa una pila cuya cabecera es "x", siendo "s" el resto de la pila

x,y,z,s representa una pila cuyo primer elemento es "x", el segundo "y", y el tercero "z". El resto de la pila se representa como "s".

Las operaciones de añadir y eliminar elementos de las pilas se realizarán siempre por la izquierda, de forma que, dada la pila cuyo contenido es:

x,s

- añadir un elemento "y" (push) a la pila la transforma en: y,x,s
- eliminar su cabecera (pop) la deja de la forma: s

- E es un registro que contiene el apuntador al contexto o entorno de evaluación en un momento dado, es decir, los valores de las variables libres en las expresiones durante su evaluación. Tradicionalmente, el contexto se representa en forma de listas, siguiendo el modelo clásico de McCarthy, a base de apuntadores entre sus elementos. Cada nivel de, definiciones en el programa, introduce una lista de valores. Así, por ejemplo un programa de la forma:

```
(. f (A)
        donde-rec  A = ....
          y        f = función (x)  ...  .)
```

maneja un contexto que en notación de listas se representa de la forma:

((x) (A f))

En este trabajo introducimos el concepto de "vector" para representar este tipo de información, de forma que el contexto es una lista de vectores cuyos elementos son los valores definidos en cada momento. Utilizando esta estructura, el contexto del programa anterior se representa

de la forma:

([x] [A f])

es decir, como una lista de dos vectores [x] y [A f], el primero de un elemento, el valor de "x", y el segundo de dos elementos, los valores de A y f.

En ambos casos, para acceder a cada uno de estos valores se genera el código:

```
LD00    para acceder a x.
LD 1,0  para acceder a A.
LD 1,1  para acceder a f.
```

cuya interpretación veremos más adelante.

- PC es un registro que contiene la dirección de la instrucción de máquina que se ejecuta

El estado del sistema en un instante determinado viene definido por el conjunto de valores de estos cuatro componentes, de forma que la ejecución de una instrucción de máquina produce un cambio de estado o transición de la forma:

s e pc d --------> s' e' pc' d'

3.1.2.- Tipos de Información utilizados

Los tipos utilizados, así como su información asociada, son los que se describen a continuación.

1) números de 16 bits (TPNUM):
   - valor del número

2) caracter (TPCAR):
   - representación ASCII del caracter

3) booleano (TPBOL):

- valor booleano Cierto o Falso

4) lista vacia (TPNIL):

- dirección de la constante NIL

5) apunte (TPAPT):

- puntero a una lista que es el contexto en el que se definió la función
- dirección donde se encuentra el código de la función.

6) receta no evaluada (TPRZ):

- puntero a una lista que es el contexto en que se creó la receta.
- dirección donde se encuentra el código de la receta.

Definimos informalmente un apunte o una receta de la forma:

```
type apunte, receta= record
                        contexto: dirección_lista;
                        código:  dirección_código;
                        end
```

A lo largo de la exposición, referenciaremos estos dos últimos tipos de información de la forma:

apt (el, pcl) y rz (el, pcl)

donde, "el" representa el contexto en que se definió la función o se creó la receta y "pcl" la dirección donde se encuentra su código.

El tipo TPRZ cambia a TPRZE (receta en evaluación) cuando se está evaluando la receta, esta situación la denotaremos como rze.

7) promesa de fichero (TPPF):

- información del estado del fichero, que normalmente será un puntero a un descriptor.

8) lista normal (TPCONS):

- tipo y dirección (o valor) de su primer elemento ("car")
- tipo y dirección (o valor) del resto de la lista ("cdr")

Definimos este tipo de información, de igual modo que lo hicimos antes para apuntes y recetas, de la forma:

```
type tpc= (TPNUM, TPCAR, TPNIL, TPBOL, TPAPT, TPPF, TPRZ, TPRZE,
                TPCONS, TPERR, TPVEC)
     apuntador= record
                case tipo: tpc of
                TPCON, TPAPT, TPRZ, TPERR, TPVEC: (dir: dirección);
                TPPF: (fic: descriptor);
                TPNUM: (num: entero);
                TPCAR: (ch: ascii);
                TPBOL: (bit: lógico)
                end;
     cons= record
            car, cdr: apuntador
           end;
```

De este modo, la lista (1.2) se representará internamente como:

```
(TPNUM  1)  (TPNUM  2)
```

y la lista (a b.1), donde a y b son caracteres como:

```
(TPCAR  a)  (TPCONS **)
(TPCAR  b)  (TPNUM  1)
```

donde "**" representa la dirección de la última celda de tipo TPCONS (b.1).

9) error (TPERR):

- primer caracter de la lista
- puntero al resto de la lista, o caracter, si se trata de una lista de dos caracteres

10) vector (TPVEC):

- número de elementos del vector
- conjunto de elementos del vector

Definimos este tipo de objeto de la forma:

```
type vector = record
              noelem: entero;
              elem : array [] of apuntador
               end
```

A lo largo de la exposición referenciaremos este tipo de información de la forma:

vec(n, m)

donde "n" representa el número de elementos del vector, y "m" su dirección.

De acuerdo con el tipo de información que puede contener, definimos ahora la pila S de la forma:

```
type tps: (TPNUM, TPCAR, TPBOL, TPCONS, TPAPT, TPNIL, TPRZ, TPERR, TPVEC)
     pila_S: array [ ] of record
                          case tp: tps of
                          TPCON, TPAPT, TPRZ,
                          TPERR, TPVEC: (dir: dirección)
                          TPNUM: (num: entero);
                          TPCAR: (ch: ascii);
                          TPBOL: (bit: lógico);
                         end;
```

El tipo de información de la pila D es distinto ya que, como hemos dicho, esta pila se utiliza para guardar el estado de la máquina cuando se llama a una función o se evalua una receta, y poder restaurar este estado una vez finalizada su evaluación. Así pues, hemos incluido nuevos tipos para representar la información que puede contener la pila D. Estos tipos son:

- R: contiene una dirección de retorno, es decir el valor del PC para seguir la ejecución del programa donde se dejó.

- E: Contiene la dirección del contexto que existía cuando se llamó a una función o se inició la evaluación de una receta.

- A: Contiene la dirección de una celda de tipo "TPCONS" cuyo primer elemento (car) es una receta en evaluación.

- D: Contiene la dirección de una celda de tipo "TPCONS" cuyo "cdr" es una receta en evaluación.

- V: Contiene la dirección de un elemento de un vector, que es una receta en evaluación.

El cometido de estos últimos tipos de información, A, D y V, es el de poder reescribir la receta en evaluación con su valor, una vez que ésta ha terminado. Se verá con más detalle su utilización en el apartado dedicado al juego de instrucciones.

Ambas pilas comparten un espacio común de memoria y tienen la misma estructura. Definimos este espacio, siguiendo la notación utilizada anteriormente para definir la pila S, de la forma que se indica a continuación:

```
type tppilas: (TPNUM, TPCAR, TPBOL, TPCONS, TPAPT, TPNIL, TPRZ, TPERR
                    R, A, D, V, E);
pilas: array [ ] of record
                        case tp: tppilas of
                        TPCON, TPAPT, TPRZ,
                        TPERR, TPVEC: (dir: dirección)
                        TPNUM: (num: entero);
                        TPCAR: (ch: ascii);
                        TPBOL: (bit: lógico);
                        R: (pc: dirección);
                        E: (contex: dirección);
                        A: (rzel: dirección);
                        D: (rze-1: dirección);
                        V: (elem: dirección);
                      end;
```

Las recetas nacen con el concepto de evaluación "retardada" de Henderson [26], concepto similar a las "suspensiones" de Friedman [17]. La idea consiste básicamente en no evaluar nada hasta que no se necesite, de esta forma podemos "suspender" la evaluación de expresiones, construyendo un objeto que llamamos "receta" y que consiste en una referencia a la expresión cuya evaluación fué suspendida (en nuestro caso, la dirección de una zona de código) y una referencia al contexto en que fué creada y por lo tanto necesario para su evaluación. Cuando durante la ejecución del programa se necesite de este valor será cuando únicamente se evalúe, sustituyéndose finalmente el objeto "receta" por su valor (resultado de su evaluación).

El tipo TPRZE, receta en curso de evaluación, es la marca de que una receta se está evaluando. Permite detectar definiciones circulares en el programa, que no son resolubles por la máquina. En el caso de un programa "monotarea", si durante la ejecución del programa, se alcanza una receta en estas condiciones, significa que para evaluar una expresión es necesario conocerla de antemano, lo cual conllevaría a un bucle infinito. La máquina detecta este hecho, avisando al usuario con un mensaje de error, como veremos más adelante. Sin embargo, cuando existen varias tareas encargadas de la ejecución del programa (programa "multitarea"), generalmente comparten

estructuras comunes, por lo que el hecho de que una de ellas encuentre una receta en evaluación no implica necesariamente una situación errónea. Puede ocurrir simplemente que deba esperar a que otra tarea acabe su evaluación para poder tomar el resultado. Es preciso entonces, añadir nuevos mecanismos para detectar la circularidad de definiciones, como veremos en el capítulo 4. En lo referente al capítulo que nos ocupa, supondremos una sóla tarea encargada de la ejecución del programa.

Las promesas de fichero nacen de los flujos (streams) propuestos por Landin [34] y posteriormente estudiados por Burge [6], y su nombre proviene de los trabajos de Friedman y Wise [19]). Un flujo es un tipo especial de función que representa una secuencia o lista infinita, función aplicable a una lista de argumentos vacía, que devuelve un par. Par, cuyo primer elemento es el primero de la lista y cuyo segundo es un flujo que representa el resto de la lista. Esto permite posponer su evaluación hasta que realmente se necesiten los elementos de la lista.

Nuestra función primitiva FICH abre un fichero secuencial que viene dado por su argumento y devuelve un par cuyo primer elemento es el primer caracter del fichero y el siguiente es una "promesa de fichero", que denotaremos como "pf()" y representa el resto del fichero. Un fichero es, para nuestro sistema, una secuencia de caracteres que no se lee inmediatamente, sino que es aplazada. Esta es otra aplicación de la llamada "evaluación retardada", a la que hicimos referencia anteriormente, donde las estructuras "suspendidas" son los ficheros mismos.

Las listas de tipo ERROR aparecen como ya en Mañas [35], como una manera de informar al usuario del tipo de error cometido, así como del entorno en el que se produjo. Un error en un sistema funcional, es simplemente una expresión que no se puede reducir.

A lo largo de este capítulo veremos, con mayor detalle, la aplicación de cada uno de los conceptos aquí expuestos.

3.2.-CODIGO LINEAL

Como ya es sabido, la elección del tipo de representación para el código a ejecutar por un computador, es de gran importancia. No tanto en cuanto al número de palabras de memoria que éste ocupa, como en cuanto al contínuo acceso que es preciso realizar a este tipo de información a lo largo de la ejecución de un programa.

Ya que, en un sistema funcional, hay que ejecutar las instrucciones de máquina de forma secuencial, cabe preguntarse por qué utilizar una representación en forma de listas, como la utilizada por Henderson [27]. Con este tipo de estructuras, es necesario un nivel más de indirección que en una representación convencional (lineal), esto es, hay que recorrer el código siguiendo apuntadores.

Pensamos, que si representamos en nuestro sistema el código de forma lineal, utilizando un registro que contenga en cada momento, la dirección de la siguiente instrucción a ejecutar, habremos obtenido, en principio, ventajas evidentes con respecto a la representación de listas.

Otro hecho a tener en cuenta es, que la representación de listas permite recolectar espacio de memoria, o lo que es lo mismo, hacer reutilizables celdas de memoria que ya no se usan. La utilización de una representación lineal obliga, sin embargo, a que el código ocupe un número fijo de palabras a lo largo de toda la ejecución del programa. Cabe preguntarse, por tanto, si realmente se recolecta una cantidad suficiéntemente grande de memoria, correspondiente a código, como para decir, que a pesar de sus inconvenientes, la representación de listas es beneficiosa, a la larga, con respecto a la lineal.

Hemos realizado una serie de medidas sobre programas de prueba, que expondremos en detalle en el capítulo 5, cuyos resultados nos permiten afirmar que, el número de celdas de memoria, correspondientes a código, liberadas tras cada recolección, es prácticamente insignificante, y que aunque en principio podría pensarse que el código se destruye a lo largo de la ejecución del programa (debido por ejemplo al abandono del código de recetas ya evaluadas), esto no es del todo cierto como podremos comprobar en el

capítulo 5.

Vistas las ventajas que nos aporta la utilización de una representación lineal para el código, y una vez sopesados sus pros y contras, hemos adoptado ésta en el sistema que se presenta en este trabajo.

Pasamos a continuación a ver el juego de instrucciones de máquina, propuesto para este sistema. Posteriormente, expondremos las optimizaciones que se han realizado a lo largo de su desarrollo, y finalmente, veremos los mensajes de error a los que pueden dar lugar durante su ejecución.

3.2.1.- JUEGO DE INSTRUCCIONES DE MÁQUINA

Presentamos aquí un juego básico de instrucciones de máquina para un sistema funcional, basado en los desarrollados por Henderson [27] y Mañas [35]. En general, supone un cambio significativo respecto a estos autores, ya que en nuestro caso, el código a ejecutar por la máquina está fijo en memoria como resultado de la utilización de una representación lineal, mientras que en aquellos, se seguía un modelo de representación de listas.

En este apartado, representaremos el contexto utilizando la notación de listas exclusivamente. Dejamos para el apartado 3.3 la exposición de los cambios que introduce la inclusión del objeto "vector" en la representación de este componente del sistema.

Definimos una instrucción como una orden elemental dada a la máquina que hace que ésta cambie de estado. Así pues, podemos ver su acción como la transición de un estado a otro, es decir, un cambio en los valores de sus componentes básicos S, E, PC y D. Por ejemplo, la acción de la instrucción LDCN para cargar un número en la pila S queda definida por la transición:

s e pc d --------> <número>,s e pc+2 d

y su efecto es cargar el valor <número> en la cima de la pila S e incrementar el registro PC para obtener la siguiente instrucción a ejecutar.

Pasamos a continuación, a describir cada una de las instrucciones de máquina existentes. A lo largo de la exposición haremos mayor hincapié en aquellas instrucciones que suponen un cambio sustancial con respecto a los modelos de otros autores. Definimos los siguientes grupos de instrucciones:

I.- Instrucciones de creación o destrucción de niveles de definiciones en el contexto
II.- Carga de la pila S con valores del contexto
III.- Carga de Constantes
IV.- Carga de Funciones y Recetas
V.- Aplicación de Funciones de usuario
VI.- Instrucciones de Salto
VII.- Construcción y destrucción de listas
VIII.- Entrada/Salida
IX.- Funciones primitivas

I.- Instrucciones de creación o destrucción de niveles de definiciones en el contexto

1.- ECONS

x,s e pc d --------> s (x.e) pc+1 d

Extiende el contexto E con una lista de valores preparada en la pila S. Esta instrucción aparece al compilar bloques de definiciones de la forma:

(. sea <expresión> $X_1 = e_1$ y $X_2 = e_2$ y .... y $X_n = e_n$.)

Con esta instrucción enriquecemos el contexto existente hasta el momento con el conjunto de expresiones e1, e2 ..., en.

2.- NCONS

s e pc d ----------> s (NIL.e) pc+1 d

Crea un nivel fictício en el contexto a la hora de evaluar bloques de definiciones recursívas de la forma:

(. sea-rec <expresión> X_1= e_1 y X_2= e_2 y .... y X_n= e_n.)

La creación de este nivel ficticio es simplemente un modo de cubrir temporalmente el hueco del contexto real que aparecerá mas tarde, compuesto por las definiciones del bloque.

3.- ERPL

x,s (NIL.e) pc d ---------> s (x.e) pc+1 d

Esta instrucción completa a la anterior. Sustituye el nivel ficticio, introducido en el contexto por NCONS, por la lista preparada en la pila S, que no será sino la lista de definiciones del bloque recurrente al que nos referimos anteriormente.

4.- ECDR

s (x.e) pc d ----------> s e pc+1 d

Elimina del contexto el último nivel de definiciones añadido. Esta instrucción aparecerá cuando se haya finalizado la evaluación de un bloque, recurrente o no, eliminando de esta forma sus definiciones e_1, e_2, ... , e_n, que no serán de ninguna utilidad fuera de este bloque. Recordemos que este nivel del contexto lo introdujeron ECONS o NCONS.

II.- Carga de la pila S con valores del contexto

5.- LD n,v

Esta es una instrucción de 3 palabras que se utiliza para acceder a

valores "x" almacenados en el contexto y cargarlos en la pila S. Los parámetros "n" y "v" indican la posición del valor al que se quiere acceder dentro del contexto, la sublista número "v" de la lista "n" del contexto. Utilizando la notación de selectores, tendríamos la siguiente formulación:

$$x = v+1\ (n+1\ (e))$$

Los casos que se pueden presentar son los siguientes:

a) Que x sea una receta no evaluada: x= rz (el,pcl).

En este caso se procede a su evaluación. El hecho de evaluar una receta se puede asemejar a la llamada a una subrutina en lenguajes de programación convencionales. Es necesario guardar el estado actual de la máquina antes de proceder a la evaluación de la receta. En principio, se deben guardar: el contexto existente en ese momento y la dirección de la instrucción siguiente a LD.

La receta se marcará como en curso de evaluación (rze) y se pasará también a la pila D junto con la estructura que la contiene. Esto es, en la pila D se guardará la dirección de una celda de tipo TPCONS, cuyo "car" es un objeto de tipo receta en evaluación y cuyo "cdr" llamaremos "b". La transición por tanto será:

si x= 1 (rz (el,pcl).b) ent

s e pc d ------> s el pcl E:e,A:(rze.b),R:pc+3,d

Como ya dijimos anteriormente, la pila D contiene tipo y dirección de la información que guarda. El hecho de guardar en D la palabra A:(rze.b) es necesario para que una vez evaluada la receta se sustituya esta por su valor. Para ello necesitaremos saber en qué lugar del contexto está, cosa que nos viene dada por la dirección de la celda (rze.b). De este modo cualquier acceso posterior a "x" encontrará no la receta sino su valor.

La evaluación de la receta consiste en ejecutar el código etiquetado con "pcl" en el contexto "el". La última instrucción del código

de la receta será la instrucción RET que veremos más adelante. Esta instrucción encontrará en la cima de S el valor de la receta y restaurará el estado de la máquina produciendo la transición:

v,s e' pc' E:e,A:(rze.b),R:pc+3, d -----> v,s e pc+3 d

y como efecto secundario sustituirá la celda (rze.b) por (v.b), es decir, sustituirá la receta por su valor.

b) Que x sea una receta en curso de evaluación: x= rze.

En este caso tendremos una definición circular irresoluble a la que hicimos referencia anteriormente, y se generará una lista de tipo error para informar al usuario del error cometido.

c) En el resto de los casos, es decir, cuando "x" es un átomo, una lista normal, un apunte o una lista de tipo error, la acción de LD es simplemente cargar su valor "v" en la pila S, es decir, vendrá definida por la transición:

s e pc d ----------> v,s e pc+3 d

6.- LD0 v

7.- LD00

Estas dos instrucciones son casos particulares de la instrucción LD n v. LD00 es el caso particular en que tanto n como v son cero, y LD0 v es el caso particular en que n es 0. Se dan por tanto los mismos casos vistos para LD n v.

III.- Carga de Constantes

8.- LDCN <número>

s e pc d ---------> <número>,s e pc+2 d

Carga un número entero en la pila S.

9.- LDZ

s e pc d ---------> 0,s e pc+1 d

Carga la constante 0 en la pila S. Este es un caso particular de la instrucción anterior.

10.- LDCC <caracter>

s e pc d ---------> <caracter>,s e pc+2 d

Carga un caracter en la pila S (su representación ASCII)

11.- LDCB <bool>

s e pc d ---------> <bool>,s e pc+2 d

Carga un valor booleano en la pila S.

12.- LDN

s e pc d ---------> NIL,s e pc+1 d

Carga la constante NIL en la pila S.

IV.- Carga de Funciones y Recetas

13.- LDF pc1

s e pc d ---------> apt(e1, pc1),s e pc+2 d

Crea un apunte y lo coloca en la pila S, esto es, crea una celda de tipo apunte que contiene el puntero al contexto en el que se definió la función "e1", así como la etiqueta o dirección "pc1" donde está compilado su código.

La realización de esta instrucción se ha hecho de tal forma que la

primera palabra de esta zona de código no es una instrucción en sí, si no una palabra que contiene el número de argumentos formales de la función. De este modo, antes de aplicar la función a una serie de argumentos reales se puede comprobar, si procede, que el número de argumentos formales coincide con el número de argumentos reales, en lugar de dejar que sean las instrucciones LD las que generen un error al intentar acceder a un argumento que no existe. Profundizaremos más en el tema a lo largo de esta exposición.

14.- LDR pcl

s e pc d ---------> rz(e, pcl),s e pc+2 d

Crea una receta y la coloca en la pila S. La creación de una receta lleva consigo la creación de un objeto de tipo "TPRZ" que contiene, como dijimos anteriormente, un puntero al contexto en el que se crea y una etiqueta o dirección donde está compilado su código.

Esta es la única ocasión en la que puede aparecer una receta en la pila S, ya que posteriores accesos a ella (ver las instrucciones LD, CAR y CDR) provocarán su evaluación.

V.- Aplicación de Funciones de usuario

15.- AP

x,v,s e pc d --------->

Esta es la instrucción que aplica funciones definidas por el usuario. AP espera encontrar en la pila S la función a aplicar y los argumentos reales a los que se aplicará esta. Pueden darse dos casos:

a) Que AP encuentre un apunte en la cima de S, es decir, x= apt (el,pcl).

En este caso, se procederá a aplicar la función a la lista "v" de argumentos reales, que estará preparada en la siguiente palabra de la pila S.

apt (el,pcl),v,s e pc d --------> s (v.el) pcl+1 E:e,R:pc+1,d

Se pasa a evaluar el código de la función, que comienza en la dirección pcl+1 (como dijimos, la dirección pcl contiene el número de argumentos reales de la función) en un contexto (v.el) que es el contexto en el que se definió la función, el, enriquecido con los parámetros reales de ésta, "v". Esta evaluación conlleva el tener que guardar el estado de la máquina, "e y pc+1", para que cuando acabe (instrucción RET) se pueda volver al punto donde se dejó el proceso de evaluación para llamar a la función.

b) Que AP encuentre un número en la cima de S.

Esto significa que se trata de la aplicación de un selector i a una lista.

```
i,v,s  e  pc  d  ------>
    si i= 0  ent ------> l(v),s  e  pc+1  d
    si i= -1 ent ------> l(v),s  e  RCD   R:pc+1, d
    si i= 1  ent ------> l(v),s  e  RCA   R:pc+1, d
    si no        ------> l(v),i,s  e  RCN   R:pc+1, d
```

Ya que, el procedimiento general de aplicación de funciones requiere previamente, construir y guardar en S la lista "v" de argumentos reales de la función, y debido a que, en el caso que nos ocupa, sólo es necesario un argumento, esto es, la lista a la que se aplica el selector, hay que deshacer la lista construida y quedarnos sólo con su primer elemento, "l(v)".

En la transición de estados hemos utilizado los nombres RCD, RCA y RCN para identificar rutinas del sistema introducidas, que se indican a continuación.

```
RCD: CDR
     RET
RCA: CAR
     RET
RDN: CDR
RCN: CNR
```

En los casos i= 1 ó i= -1, AP podría calcular ella misma el "car" o el "cdr" de la lista, pero ésta no es una tarea simple (ver instrucciones CAR y CDR) ya que podría encontrarse con una receta, una promesa de fichero, etc. Por tanto, es preferible descargar a AP de esta tarea y dejar que otras instrucciones existentes para ese fin la realicen. Con mayor motivo no debe AP hacer estos cálculos, si i > 1 ó i < -1, ya que AP tendría que recorrer una lista y en este recorrido podría encontrar recetas, que debería evaluar para seguir su recorrido, pudiendo a su vez aparecer nuevos selectores, etc. En este caso se utilizará la instrucción CNR que realizará este trabajo, descargando a AP de tareas que no le son propias.

16.- VRF n

La misión de esta instrucción es la de comprobar, en tiempo de ejecución, que el número de argumentos reales coincide con el número de argumentos formales de la función que se va a aplicar o que el número de argumentos a los que se aplica un selector es 1. El compilador sólo deberá generar esta instrucción cuando no sea fácil comprobar este hecho en tiempo de compilación. Esto sucede en el caso de funciones cuyos argumentos son a su vez funciones o que devuelven como resultado una función. Siempre que aparece esta instrucción en un programa, va seguida de la instrucción AP o de sus optimizaciones APN y APNE que veremos más adelante.

Se pueden dar para esta instrucción los mismos casos que se daban en AP.

x,v,s e pc d --------->

a) Que encuentre un apunte en la pila S; x= apt (el,pcl).

En este caso, VRF debe comprobar que el número "n" de argumentos reales de la función, coincide con el número "n'" de argumentos formales. Así pues, la transición que se producirá en este caso será:

si n= n' ent --------> x,v,s e pc+2 d
sino "error"
donde n' = "contenido de la dirección pc1"

En el caso n = n' ,VRF se convierte en una instrucción de "no operación". En caso contrario, se genera una lista de tipo error, que indica si el número de argumentos era insuficiente o excesivo.

b) Que encuentre un número.

Recordemos que éste era el caso de la aplicación de un selector a una lista. En este caso, se debe verificar que el número de argumentos al que se aplica el selector es 1, siendo este argumento, la lista a la que se quiere aplicar. Por tanto:

si n = 1 ent ---------> i,v,s e pc+2 d
sino "error"

17.- RET

Su misión es restaurar en la máquina el estado en el que se encontraba cuando se produjo el salto a la zona de código de una función o una receta. Se pueden dar dos casos:

a) Que RET encuentre vacía la pila D:

s e pc vacía --------> s e pc vacía

Este sería el retorno final, es decir, el fin de ejecución del programa. La máquina quedaría conceptualmente en un bucle infinito. Equivale a la clásica instrucción HALT.

b) Que encuentre algo en la pila D.

Llamando d1 a la primera palabra de la pila D y d2 al resto de la pila, definimos la acción de RET de la siguiente forma:

```
s   e   pc   d --------> s   NIL   pc   d ---------->

bucle
      sea d= d1,d2 y s= x,s1
      cond
            si d1= R:pc1     ent ------> s  e  pc1   d2
                                    exit
            si d1= A:(rze.b) ent ------> s  e  pc    d2
                                    y (rze.b) ---> (x.b)
            si d1= D:(b.rze) ent ------> s  e  pc    d2
                                    y (b.rze) ---> (b.x)
            sino    -- d1= E:e1
                        -------> s  e1  pc   d2
endbucle
```

En resumen, RET se realiza como un bucle de búsqueda de una dirección de retorno. Si se encuentra en la pila D una palabra de tipo A o D, reescribe una receta en evaluación con su valor recién calculado y continua la búsqueda. Si encuentra una palabra de tipo E, restaura el contexto y continua la búsqueda. Saldrá del bucle únicamente cuando encuentre una dirección de retorno, es decir una palabra de tipo R.

VI.- Instrucciones de Salto

18.- JF pc1

```
x,s   e   pc   d -------->
        si x= "falso" ent -------> s   e   pc1   d
        si no  --------> s   e   pc+2   d
```

Instrucción de salto condicional. Si encuentra en la cima de la pila S el valor booleano "falso", se produce un salto a ejecutar la instruc-

ción cuya dirección es pc1, si no, se sigue en secuencia. Caso de no encontrar un valor booleano, se producirá un error

19.- J pc1

s e pc d --------> s e pc1 d

Instrucción de salto incondicional a la dirección pc1.

Las instrucciones de salto aparecen como consecuencia de la compilación de expresiones condicionales simples :

(si p ent ec si no ef)

o múltiples:

(cond si p1 ent e_1
si p2 ent e_2
.....
si no e_n)

VII.- Construcción y destrucción de listas

20.- CAR

(x.b),s e pc d ---------->

Acceso y carga en S del "car" de una lista normal de tipo CONS. Se pueden dar los siguientes casos:

a) Que el "car" de la lista sea una receta: x= rz (e1, pc1).

En este caso se procederá a la evaluación de la receta:

----------> s e1 pc1 E:e,A:(rze.b),R:pc+1,d

La transición es idéntica a la vista para LD, ya que la situación es la misma: acceso a una receta.

b) Que sea una receta en evaluación: x= rze, en cuyo caso se trata de una

definición circular irresoluble, generándose el correspondiente error.

c) Que sea un átomo, lista, apunte, error, etc., en cuyo caso se carga sin más en la cima de la pila S:

----------> x,s e pc+1 d

Pudiera darse el caso de que en S no hubiera una lista de tipo CONS. En este caso se produciría un error.

21.- CDR

(b.x),s e pc d ----------->

Acceso y carga en S del "cdr" de una lista normal de tipo CONS que está en la cima de la pila S. Se pueden dar los mismos casos que vimos para CAR y su tratamiento es análogo.

El caso a) producirá la transición:

----------> s el pc1 E:e,D:(b.rze),R:pc+1,d

Además de los casos vistos para CAR, se puede dar otro caso:

e) Que x sea una promesa de fichero: x= pf().

En este caso, la acción de CDR es "leer el siguiente caracter" del fichero que indica pf(). Como dijimos en el apartado 3.1.2, una celda de este tipo contiene información acerca del estado del fichero, y normalmente será un puntero a un descriptor. La transición que tendrá lugar será:

----------> (a.pf'()),s e pc+1 d

donde, a representa el caracter leido y pf'() una nueva promesa de fichero que reflejará el nuevo estado del fichero, que no es sino el resto de la información que contiene.

Además del efecto de esta transición, tenemos el efecto de reescritura de la celda original (b.pf()) con la información obtenida del fichero, es decir,

(b.pf()) -------> (b a.pf'())

En caso de detectarse el fin del fichero, que no sería mas que el fin de una lista de caracteres, tendríamos la transición:

----------> NIL,s e pc+1 d
y (b.pf()) -----> (b.NIL)

22.- CNR

v,i,s e pc d --------->

si i> 1 ent ------>	v,i-1,s	e	RDN	d
si i= 1 ent ------>	v,s	e	RCA	d
si i= -1 ent ------>	v,s	e	RCD	d
si i< -1 ent ------>	v,i+1,s	e	RDN	d

Esta es la instrucción encargada de aplicar el selector i a la lista v. Por el mismo motivo que comentábamos en la instrucción AP, CNR no evalúa directamente i(v), sino que llama a las rutinas del sistema, delegando en las instrucciones de estas rutinas la compleja tarea de evaluar recetas, promesas de fichero, etc.

23.- CONS

a,b,s e pc d ---------> (a.b),s e pc+1 d

Función primitiva de construcción de listas que por su simplicidad creemos no necesita comentarios.

VIII.- Entrada/Salida

24.- FICH

id,s e pc d ----------> (a.pf()),s e pc+1 d

Abre el fichero cuyo nombre viene dado por la lista de caracteres "id", resultado de la evaluación de su argumento, y lee su primer caracter, que llamamos "a", poniendo sobre la pila S una lista formada por este caracter y una promesa de fichero que indica el nuevo estado del fichero o resto del flujo de caracteres que contiene. Así pues un fichero es considerado como un flujo o secuencia de caracteres.

El fichero podría estar vacío, en cuyo caso se produce la condición de fin de fichero:

----------> NIL,s e pc+1 d

Caso de no existir el fichero, o no ser posible su apertura, se genera el correspondiente error.

25.- IMP

Esta instrucción es la encargada de imprimir el resultado, obtenido al reducir el programa. Imprime, por tanto, átomos y errores. En su ejecución, IMP puede encontrar una lista y por tanto deberá: calcular el "car" de la lista, imprimirlo, calcular el "cdr" e imprimirlo. Este comportamiento viene reflejado en la siguiente transición:

x,s e pc d ---------->
si x= "lista de tipo CONS" ent
----------> x,x,s e RIC E:e,R:pc+1,d

siendo RIC la rutina del sistema de impresión de listas:

```
RIC: CAR
     IMP
     CDR
     IMP
     RET
```

Como se puede ver, se duplica la lista x en la pila, esto es debido a que CAR consume su argumento y éste es necesario para calcular posteriormente CDR.

Si x es una lista de tipo error, basta con aplanarla e imprimirla, de forma que si x= (x1.x2) entonces:

```
------------> x1,x2,s    e    RIE    E:e,R:pc+1,d
```

donde RIE es la rutina del sistema de impresión de listas de tipo error:

```
RIE: IMP
RII: IMP
     RET
```

Si x es un átomo, número, caracter etc., se imprime como tal, representándose los valores booleanos como C y F. Esta definición de IMP es un modo de resolver el algoritmo recursivo de escritura de una lista, en base a llamadas a una rutina del sistema.

A este modo de funcionamiento se le conoce como evaluación dirigida por la demanda de resultados. Cuando IMP se ejecuta para imprimir "algo", es únicamente cuando se comienza la evaluación de este "algo".

IX.- Funciones Primitivas

- Predicados

26.- ATM

x,s e pc d --------> bool,s e pc+1 d

Devuelve un valor booleano que será Cierto o Falso dependiendo de que x sea o no un átomo.

27.- NUL

x,s e pc d --------> bool, s e pc+1 d

Igual a la anterior, comprobando si x es el átomo NIL o no.

- Aritméticas

28.- ADD

b,a,s e pc d ------> a+b,s e pc+1 d

29.- SUB

b,a,s e pc d ------> a-b,s e pc+1 d

30.- MUL

b,a,s e pc d ------> a*b,s e pc+1 d

31.- DIV

b,a,s e pc d ------> a/b,s e pc+1 d

32.- MOD

b,a,s e pc d ------> a mod b,s e pc+1 d

33.- NEG

a,s e pc d ------> -a,s e pc+1 d

- Lógicas

34.- AND

b,a,s e pc d ------> a and b,s e pc+1 d

35.- OR

b,a,s e pc d ------> a or b,s e pc+1 d

36.- XOR

b,a,s e pc d ------> a xor b,s e pc+1 d

37.- NOT

a,s e pc d ------> not a,s e pc+1 d

- Comparación

38.- EQ

b,a,s e pc d ------> a=b,s e pc+1 d

39.- NE

b,a,s e pc d ------> a<>b,s e pc+1 d

40.- LT

b,a,s e pc d ------> a<b,s e pc+1 d

41.- LE

b,a,s e pc d ------> a<=b,s e pc+1 d

42.- GT

b,a,s e pc d ------> a>b,s e pc+1 d

43.- GE

b,a,s e pc d ------> a>=b,s e pc+1 d

La máquina parte del estado inicial:

vacía NIL pc R: RII

la pila S vacía, un contexto sin definiciones, en el contador de programa la dirección de la primera instrucción de máquina del programa y una pila D que guarda una dirección de retorno, que no es más que la dirección del código a ejecutar una vez reducido el código del programa:

```
RII: IMP
     RET
```

y que es el encargado de arrancar las evaluaciones necesarias para la impresión del resultado.

3.2.2.- OPTIMIZACIONES

Dentro del juego de instrucciones presentado en el apartado anterior, son varias las instrucciones que guardan el estado actual de la máquina en la pila D, produciendo en esta un cambio de la forma:

```
d --------> E: e, R: pc, d
```

Pero muchas veces esto no es necesario. Esto sucede cuando este estado no se emplea tras su restauración, es decir, cuando la instrucción que lo guarda va seguida de otra que restaura otro estado. El caso más clásico es el de una instrucción AP de aplicación de una función de usuario (equivalente a una llamada a subrutina) seguida de una RET de retorno, denominado "recursión por la cola" (tail recursion).

De las instrucciones expuestas son varias las que guardan el estado actual, estas son:

LD, LD0 y LD00	ante una receta a evaluar
AP	
CAR y CDR	ante una receta a evaluar
IMP	para imprimir listas

En tiempo de ejecución podría verificarse si la siguiente instrucción es RET y, de ser así, no guardar el estado en la pila D. Alternativamente, en tiempo de compilación podría verificarse si la siguiente instruc-

ción es RET, y de ser así, generar una versión especial de la instrucción que no guarde nada en esta pila. Respectívamente:

LDRET, LDORET, LDOORET, APRET, CARRET, CDRRET e IMPRET

La segunda alternatíva incrementa el número de instrucciones del computador, pero a cambio reduce el tiempo de ejecución y el volumen del programa. Ambas alternatívas reducen la carga de la pila D ya que no guardan información de estado innecesaria.

Todas las instrucciones anteriores tienen un comportamiento condicionado a los datos que encuentren en ejecución. Por ejemplo, LD puede encontrar un valor en el contexto y no necesitar guardar nada en la pila D. Pero también puede encontrar una receta y entonces tiende a guardar información de estado en la pila D. AP puede encontrarse el número 0 o puede encontarse un apunte, actuando en consecuencia. Resumiendo, todas las instrucciones anteriores pueden no necesitar optimización.

Para cuando no la necesitan, la primera alternatíva hace que la máquina pase al RET y lo ejecute sin más sofisticación. La segunda alternatíva permite eliminar el RET siempre y cuando la propia instrucción haga sus funciones, esto es, saque información de la pila D (véase RET). Con esta condición se ahorra una instrucción a cambio de complicar otras instrucciones. Hemos optado por la segunda alternatíva eliminando efectívamente la instrucción RET.

A veces no es posible optimizar tanto, pero pueden encontrarse mecanismos más débiles, como es el de guardar tan solo medio estado, la dirección de retorno, olvidando la información del contexto. Esto es así cuando el código que sigue a la instrucción afectada y hasta llegar a la siguiente restauración (RET) no hace referencia al contexto E.

La única instrucción que recupera un contexto de la pila es RET y las instrucciones que utilizan E son:

ECONS, NCONS, ERPL, ECDR
LD, LD0, LD00
LDF, LDR

El compilador, ante una instrucción que tiende a guardar el contexto, puede recorrer el código que la sigue hasta llegar a un RET o a una instrucción con RET incorporado (CNR y las ...RET del apartado anterior). Si en este recorrido no encuentra ninguna instrucción del conjunto que utiliza el contexto, reseñado más arriba, entonces estamos en condiciones de ahorrarnos el tener que guardar el contexto E. Para lograrlo generaremos alguna de las instrucciones:

LDNE, LDONE, LDOONE, APNE, CARNE, CDRNE e IMPNE

El objeto de esta optimización es doble. Primero, evitar cargar la pila D con datos inútiles. Y segundo, eliminar los contextos en cuanto no sirvan. Por esta razón, sólo se reserva un contexto si va a seguir siendo utilizado. Por esta misma razón, RET elimina el contexto, salvo que tenga que recuperar uno de la pila D.

Algunas rutinas del sistema cambian inmediatamente de la forma:

```
RCD: CDRRET

RCA: CARRET

RDN: CDRNE
RCN: CNR

RIC: CARNE
     IMPNE
     CDRNE
     IMPRET
```

RIE: IMPNE
IMPRET

Otro tipo de optimización introducida, que creemos conveniente señalar es la referente a las expresiones condicionales. El código generado para estas expresiones es generalmente:

```
    ... evaluar la condición
    JF 11
    ... evaluar la rama si cierto
    J 12
11: ... evaluar la rama si falso
12: ... seguir
```

Si la instrucción JF encuentra, en la pila S, algo diferente a los valores booleanos Cierto o Falso, ya que no sabe si cargar en S lo cierto o lo falso, carga un error indicativo de esta situación. JF debe entonces saltar a 12. Esto es posible saltando a 11-2 (instrucción J 12) para a continuación saltar a 12, manteniendo coherente de este modo la pila S.

Una optimización clásica del compilador es detectar una instrucción RET en 12 y generar:

```
    ... evaluar la condición
    JF 11
    ... evaluar la rama si cierto
    RET
11: ... evaluar la rama si falso
    RET
```

El problema es entonces, que RET ocupa una palabra mientras que J 12 ocupa dos. La solución más simple es que el compilador desprecie la palabra anterior a la etiquetada con 11. Así, si JF salta a 11-2 y encuentra RET la evaluación continúa correctamente.

Veamos con un ejemplo el efecto combinado de las optimizaciones expuestas.

Sea la función:

```
facaux = función (n, p)
          si n = 0 ent  p
          si no   (n-1, n*p)
```

Para esta definición se genera el código:

```
          LDZ
          LDOO              -- referencia al contexto (n)
          EQ
          JF rf
          LDORET 1          -- referencia al contexto (p)
                            -- optimización: 12= RET
rf:       LDN
          LDR s2
          CONS
          LDR s1
          CONS
          LDNE 1, ?         -- referencia al contexto (facaux)
                            -- optimización: no guardar contexto
          APRET
s1:       LDCN 1
          LDOONE            -- n
          SUB               -- no usa contexto
          RET

s2:       LDO 1             -- p
          LDOONE            -- n
          MUL               -- no usa contexto
          RET
```

En el capítulo dedicado a evaluaciones veremos las consecuencias que estas optimizaciones tienen sobre el rendimiento del sistema.

3.2.3.- ERRORES EN EJECUCION

Como dijimos anteriormente, en un sistema funcional los errores son simplemente, expresiones que no se pueden reducir. El objetívo planteado aquí, es dar una respuesta informatíva al usuario, respuesta que le de a conocer, el tipo de error cometido y el entorno en que se cometió. Para ello, nos hemos basado en la utilización de listas de tipo ERROR, introducidas en (Mañas [35]), consiguiendo finalmente un resultado que permite al usuario localizar sin problemas su error. Nuestra propia experiencia nos ha demostrado la eficacia de este tipo de respuesta a errores de programación, en la depuración de programas escritos en un lenguaje funcional.

En la exposición del juego de instrucciones, nos referimos ya a los errores que se pueden encontrar durante la ejecución de un programa. Los tipos de error existentes aparecen en el apéndice 1, en relación con las instrucciones de máquina que los detectan. A continuación trataremos estos en el mismo orden en que aparecieron en aquella sección. Para evitar redundancias, no repetiremos los casos para las versiones optimizadas, así, lo que se expone para LD es igualmente, válido para LDRET y LDNE.

El primer tipo de error considerado se produce cuando las instrucciones LD, LD0, LD00, CAR y CDR encuentran una receta en evaluación (rze). Esto indica que la máquina está evaluando esa receta, y que para conocer su valor necesita conocerla previamente. Aclaremos esto con un ejemplo. Sea la expresión:

```
(. X
  donde-rec X = 1 + X  .)
```

Al intentar evaluar X nos encontrarmos con la expresión 1+X. Para evaluar 1+X se debe conocer previamente el valor de X y eso es precisamente lo que se está intentando evaluar. La respuesta a esta situación es la generación de una lista de caracteres, de tipo ERROR, de la forma:

```
"*dci*"
```

que se lee " definición circular inevaluable". Esta lista se carga, en la

pila S, en lugar del valor deseado. En todo lo demás, la instrucción opera como si hubiera localizado un valor.

Otro tipo de error aparece cuando la instrucción AP, que espera encontrar, en la pila S, un apunte o un selector a aplicar sobre unos argumentos, no encuentra ninguna de estas cosas. Por ejemplo, la expresión:

```
(. alfa (2)
  donde alfa = 1:2:3 .)
```

es claramente errónea, ya que una lista, alfa, no es aplicable a nada. Ante una situación de este tipo, la máquina responde, cargando en la pila S la lista tipo ERROR:

```
"(x *nea*)"
```

que se lee "x no es aplicable". En nuestro caso:

```
"((1 2.3) *nea*)"
```

es decir, "la lista (1 2.3) no es aplicable".

En cualquier caso, AP consume la pretendida función y sus argumentos reales, ocupando la lista de error la posición que hubiera correspondido al resultado de tal aplicación, caso de haberse realizado con éxito.

Siguiendo con la aplicación de funciones de usuario, veamos los errores detectados por la instrucción VRF, que como dijimos en su momento, verifica que coincidan en número los argumentos reales y formales de una función. Esta instrucción podría encontrarse con la misma situación vista para AP, es decir, tener que verificar los argumentos de algo, que no es ni un apunte ni un selector. En este caso, VRF no hace nada, dejando a AP la labor de generar el error del que hablábamos más arriba.

Otro error con el que se podría encontrar VRF, es que no se cumpliera la condición "número de argumentos reales igual a número de argumentos formales". Si el número de argumentos reales es insuficiente, se genera

y carga en la pila S la lista de tipo error:

"x *iar*"

que se lee "x tiene insuficientes argumentos reales"

Por el contrario, si es excesivo, se genera la lista:

"x *ear*"

Estas listas de error reemplazan a la función o selector a aplicar, respetando los argumentos reales. Veamos como ejemplo un caso típico de error detectado por VRF. Sea la expresión:

```
(. map (g, lis)
  donde-rec
        map = función (f, e=(el.e2))
                si nulo (e) ent NIL sino
                        f (el) : map (f, e2)
  y     g   = función (a, b)  a + b
  y     lis = 1:2:3
```

El error está cuando "map" intenta aplicar "f" a un solo argumento. Pero "f" es "g", función de dos argumentos, luego hay un número insuficiente de argumentos reales y VRF genera el error:

"$A *iar*"

donde $A indica que se intenta aplicar un apunte.

Tras la verificación, AP intentará aplicar un objeto de tipo error, propagándose éste error y dando lugar a la lista:

($A *iar* *nea*)

Otra instrucción en la que se puede presentar un error, es la instrucción de bifurcación condicional JF, que espera encontrar, en la pila S,

un valor booleano CIERTO o FALSO. Si no fuera así, como por ejemplo en la expresión:

(. si "a" ent 0 sino 1 .)

donde encontraría el caracter "a" en lugar del valor booleano que espera, no podría decidir la rama a ejecutar. Para informar de esto, se carga en S el mensaje:

"(SI x)"

donde "x" es el valor testeado, en nuestro caso, el caracter "a", por lo que, la lista de error generada será:

"(SI a)"

Pueden producise también situaciones de error, al intentar aplicar un selector, por ejemplo, al ejecutar una instrucción CAR o CDR, se podría dar el caso de encontrar, en S, un argumento que no fuese una lista de tipo "cons", que es únicamente a lo que se puede aplicar un selector. Ante esta situación, el resultado sería el mensaje:

"(i x)"

donde, "i" es el selector a aplicar (1 o -1), y "x" el argumento real al que se intentó aplicar. Por ejemplo, la expresión:

1 ("a")

es errónea, ya que no se puede evaluar el CAR de un átomo. El mensaje generado sería por tanto:

"(1 a)"

En general, la expresión i (x), que aplica el selector "i" a una lista "x", se descompone durante su ejecución en una secuencia de selectores 1 y -1, con lo cual se irá transmitiendo el error de unos a otros. Por

ejemplo, la expresión:

3 ("a")

generaría:

"(1 (-1 (-1 a)))"

Una de las instrucciones de entrada/salida que puede dar lugar a errores durante su ejecución, es la instrucción FICH. Esta instrucción puede encontrarse con que el fichero resultante de la evaluación de su argumento, o bien no existe, o no se puede abrir por algún motivo, por ejemplo, porque sea sólamente de escritura. Se informará de este hecho de la forma:

"(FICH exp)"

donde "exp" es el argumento que debería haber sido el nombre de un fichero existente en el sistema.

Otro tipo de error que puede aparecer durante la ejecución de un programa está relacionado con los predicados "átomo" y "nulo". Las instrucciones asociadas a estos predicados, ATM y NUL, pueden encontrarse con una lista de tipo ERROR, generada por expresiones más internas. Ante esta situación se responde con el mensaje

"(ATM x)" o "(NUL x)"

que se carga en la pila S, donde "x" es la lista que encontró la instrucción en la cima de S. Esto ocurre, por ejemplo, en la expresión:

```
(. si atm (X) ent 1
        sino 0
   donde-rec X= 1 + X  .)
```

Como vimos anteriormente, la definición X= X+1 no se puede reducir. Al intentar evaluar X, para comprobar si es o no un átomo, resultará la lista de ERROR:

"(+ 1 (*dci*))"

La instrucción ATM encontrará esta lista, generando:

"(ATM (+ 1 (*dci*)))"

en lugar de un valor booleano. Por último, al intentar decidir JF, la rama a evaluar (la de CIERTO o la de FALSO), quedará en la pila S, la lista:

"(SI (ATM (+ 1 (*dci*))))"

que será el resultado del programa.

En este caso, ATM podría haber realizado su operación, resultando el valor FALSO, ya que una lista de tipo ERROR no es un átomo, pero de hacerlo así, se perdería el rastro del error, y el usuario no tendría conocimiento de éste. Los errores siempre se deben propagar desde las expresiones más internas a las más externas, para hacer posible su localización.

Por último, consideraremos los posibles errores que pueden aparecer en las operaciones aritméticas y lógicas. Estas operaciones tienen una serie de restricciones en cuanto al tipo de operandos que aceptan. Para todas ellas, los operandos deben ser átomos, pero además, deben ser números para las aritméticas. Cualquier otra combinación de operandos produciría un error. Por ejemplo para la suma:

"(+ x y)"

donde, "x" e "y", son el primer y segundo operando respectivamente. Además, las operaciones /, * y MOD, comprueban que su segundo operando no sea cero, generando un error cuando lo sea.

Los operadores lógicos AND, OR, XOR y NOT sólo están definidos sobre operandos booleanos y devuelven valores de este tipo. La información sobre errores, en este tipo de operaciones, es similar a la vista para las operaciones aritméticas:

"(AND x y)"

"(OR x y)" ..... etc.

A lo largo de esta exposición, hemos podido comprobar cómo se transmiten los errores, de los niveles de expresiones más internos a los más externos. Debido a este hecho, las listas de tipo ERROR, pueden incrementarse de forma explosíva. Al usuario le bastará, con que aparezca el lugar exacto donde ha cometido el error, y unos pocos niveles más de anidamiento, para poder localizarlo en su programa fuente. Por otra parte, si pensamos que cada caracter ocupa una celda de memoria y revisamos los mensajes de error generados por las instrucciones, nos damos cuenta de la gran cantidad de espacio que consumen las listas de tipo ERROR.

Ante estos hechos, tomamos la decisión de generar un número determinado de listas de tipo ERROR, por ejemplo, 10. Una vez se ha generado este número de errores se aborta la ejecución, ya que no hay mucho más que hacer, sólamente imprimirlos. Al final del ciclo de cada instrucción, si la cuenta de listas de tipo ERROR es mayor que la cota fijada se realiza la transición:

x,s e pc d -------> x e RII <u>vacía</u>

es decir, se elimina el código que queda por ejecutar, la información de estado almacenada en la pila D (las reducciones pendientes a niveles más externos) y la información existente en la pila S, exceptuando su cabecera, que será el resultado obtenido hasta ese momento (una lista de tipo ERROR) y se imprime este resultado.

Cuando existen varias tareas en el sistema, esta finalización no puede ser tan tajante, ya que es posible que queden abandonadas recetas, marcadas como "en curso de evaluación", de las que dependen otras tareas, pudiendo quedar estas últimas bloqueadas para siempre. Trataremos en detalle este tema en el capítulo siguiente.

3.3.- CONTEXTO LINEAL

Definimos anteriormente el contexto o entorno de evaluación, como el conjunto de valores definidos en cada momento, o conjunto de valores de las variables libres en las expresiones. Utilizamos allí, la representación empleada por Henderson [27], en la que el contexto es una lista de listas de valores, que a su vez pueden ser listas.

La forma de acceder a un valor es, localizarlo según la posición que ocupa en esta estructura, es decir, mediante instrucciones "LD n v", cuyo cometido es localizar la sublista "v" de la sublista "n" del contexto. Esta localización debe ser totalmente secuencial o lo que es lo mismo, hay que recorrer la cadena de punteros que llevan a la sublista "n" y una vez aquí, recorrer la cadena de punteros que llevan a la sublista "v".

De acuerdo con nuestra experiencia en el manejo de estructuras lineales tales como vectores, no parece muy adecuado el uso de listas para este tipo de acceso, ya que "n" y "v" parecen mas bien índices que pudieran utilizarse para acceder directamente a una posición de memoria.

Tomando como base esta idea, y previo estudio de representaciones alternativas como la sugerida por Keller [31], planteamos una representación que, si bien es conceptualmente igual a la existente, permite un acceso más rápido a los valores del contexto. Consiste en representar este conjunto de valores como una lista de vectores, donde cada elemento del vector puede ser a su vez una lista. De esta forma, lo que antes representábamos como:

$$((e_{00}\ e_{01}\ \cdots\ e_{0n})\ (e_{10}\ e_{11}\ \cdots\ e_{1m})\ \cdots\ )$$

lo representaremos ahora de la forma:

$$([e_{00}\ e_{01}\ \cdots\ e_{0n}]\ [e_{10}\ e_{11}\ \cdots\ e_{1m}]\ \cdots\ )$$

donde cada conjunto de elementos encerrado entre corchetes es un vector o conjunto de objetos almacenados en posiciones contíguas de memoria.

3.3.1.- DEFINICION DEL TIPO VECTOR

Para materializar la idea arriba mencionada, se ha añadido un nuevo tipo de objeto, que denominamos "vector", al conjunto de los tradicionales en un sistema funcional. Definimos este nuevo tipo de la forma:

```
type vector = record
              noelem: entero;    (* número de elementos del vector *)
              elem: array [] of apuntador    (* conjunto de elementos
                                                del vector *)
              end;
```

Cada elemento de un vector puede ser un átomo (número, caracter, etc.) o una lista (la definición de una función, de un fichero o de una receta)

A la vista de esta definición, se comprueba que se ha obtenido una ventaja en lo referente a ocupación de memoria, ya que:

- Los elementos que componen los vectores son de la forma:

  tipo + información

- Los elementos utilizados para representar listas son de la forma:

  tipo del "car" + información del "car" +
  tipo del "cdr" + información del "cdr"

Las estructuras del lenguaje que introducen valores en el contexto son los bloques, no recurrentes y recurrentes, que introducen definiciones y las aplicaciones de funciones de usuario, que introducen argumentos reales. Para estas estructuras, el compilador debe generar un código tal que en ejecución introduzca en el contexto un objeto de tipo vector, compuesto por las definiciones o argumentos, almacenados en posiciones contiguas de memoria.

3.3.2.- DETALLES DE COMPILACIÓN

Compilar una expresión "e" es generar el código necesario para ejecutarla, en un contexto "n", que representa el conjunto de nombres de las variables definidas en ese momento. Esto lo denotaremos como:

comp (e, n)

Proponemos los siguientes mecanismos para la generación de código:

a) Bloques no recurrentes

comp ((sea e x_1= e_1 y .... y x_m= e_m), n) =

```
REQ m+1          -- petición de "m+1" celdas consecutívas
                 -- de memoria
                 -- para formar un vector y carga en la primera
                 -- el número de elementos del vector, "m".
comp (e1, n)     -- código de la primera definición
STR 1            -- almacenar la primera definición en la segunda
                 -- de estas celdas
....
....
comp (em, n)     -- código de la última definición
STR m            -- almacenar la última definición en la última de
                 -- las celdas
ECONS            -- añadir el vector de definiciones creado al
                 -- contexto ya existente
comp (e, n')     -- compilar "e" en el nuevo contexto n'
                 -- n' = [x1 ... xm] : n
ECDR             -- eliminar las definiciones introducidas, una vez
                 -- evaluada la expresión "e".
```

b) Bloques recurrentes.

comp ((e donde-rec x_1= e_1 y ... y x_m= e_m), n) =

```
NCONS           -- crea un nivel fictício en el contexto, al que
                -- llamaremos "vector nulo" y que representaremos
                -- como [ ].
REQ m+1
comp (e1, n')   -- compila las definiciones en el nuevo contexto
                -- n' tal que n' = [ ] : n
STR 1
....
....
comp (em, n')
STR m
ERPL            -- reemplaza  el nivel  fictício  por el vector de
                -- definiciones creado: [ ] --->[x1 ... xm]
                -- n' = [x1 ... xm] : n
comp (e, n')    -- compila "e" en el contexto real de nombres
ECDR
```

c) Aplicación de funciones de usuario.

comp ((e(e_1 e_2 .... e_m), n) =

```
REQ m+1
comp (e1, n)    -- creación del vector cuyas componentes son los
                -- parámetros reales de la función
STR 1
...
...
comp (em, n)
STR m
comp (e, n)     -- operador
AP              -- aplica el operador a los argumentos
                -- dispuestos en el vector
```

Como se puede comprobar, esto exige introducir dos nuevas instrucciones, REQ n y STR i, en el conjunto definido en el apartado anterior, así como modificar algunas de las ya definidas.

3.3.3.- MODIFICACIÓN DEL JUEGO DE INSTRUCCIONES

Comenzaremos definiendo las nuevas instrucciones introducidas, en base a las transiciones que producen en el sistema.

60.- REQ n

Pide "n" celdas consecutivas de memoria para formar un vector, y carga en la pila S un objeto de tipo TPVEC, en principio vacío. Queda definida por la transición:

s e pc d ------> vec(n,m), s e pc+2 d

donde "n" es el número de elementos del vector y "m" la dirección donde está almacenado el vector.

61.- STR i

Almacena una definición o valor, previamente preparado en la cima de la pila S, en la posición correspondiente dentro del vector de definiciones o valores al que pertenece. Queda definida por la transición:

x, vec(n,m), s e pc d -----> vec(n,m),s e pc+2 d

y tiene como efecto almacenar "x" en la celda de memoria cuya dirección es (m+i).

No se pueden dar situaciones erróneas en estas instrucciones. La única situación problemática, ocurriría en el caso de que REQ se encontrara con que no hay suficiente espacio libre en la memoria como para que pueda ser llevada a cabo su ejecución. Esto daría lugar a una recolección de celdas inútiles con objeto de liberar espacio de memoria. Si tras esta recolección hay espacio suficiente, REQ se ejecutaría y si no lo hubiera se

produciría un error máquina, cuya consecuencia sería la de abortar la ejecución sin más, ya que no hay posibilidad de seguir.

Las instrucciones que exigen una modificación, al introducir esta nueva forma de representar el contexto, son las que indicamos a continuación:

Instrucciones de creación o destrucción de niveles de definiciones en el contexto

1.- ECONS

vec(n,m),s e pc d -----> s (vec(n,m).e) pc+1 d

Extiende el contexto con un vector de valores, preparado en la pila S.

2.- NCONS

s e pc d -----> s (vec(0,NIL).e) pc+1 d

Crea un nivel fictício en el contexto, nivel que representamos mediante el vector nulo [ ] de 0 elementos y de dirección la constante NIL.

3.- ERPL

vec(n,m),s (vec(0,NIL).e) pc d
-----> s (vec(n,m).e) pc+1 d

Sustituye el nivel fictício, introducido por NCONS, por el vector de definiciones preparado en la pila S.

4.- ECDR

s (vec(n,m).e) pc d -----> s e pc+1 d

Elimina del contexto el último vector de definiciones añadido, una

vez evaluado un bloque de definiciones, recurrente o no, como vimos en la generación del código para estas expresiones.

Instrucciones de carga de S con valores del contexto

Estas instrucciones permiten, como ya dijimos, localizar un valor dentro del contexto, a partir de sus parámetros "n" y "v". Para localizar tal valor es preciso:

a) Acceder al vector n-ésimo de los que componen el contexto "e". Ya que "e" es una lista, utilizaremos la notación de selectores, de forma que:

(n+1) (e)

es el vector buscado, "vec(n,m)".

b) Una vez obtenida la dirección del vector, "m", basta hacer una suma, m+v+1, para localizar de forma directa el elemento deseado.

De este modo, sólo hay que hacer una búsqueda secuencial para obtener el vector adecuado, ya que el acceso un elemento de un vector, dada su dirección de base y su índice, es directo. Con ello, hemos obtenido una mejora sustancial en lo que se refiere a tiempo de acceso, que se estudia en detalle en el capítulo de evaluaciones.

La instrucción "LD n v", queda definida de la forma:

```
sea  x el elemento "v" del vector "vec (n,m).
si x= rz (e1, pc1)
   ent
  s   e   pc   d   -----> s   e1   pc1   E:e, V:(m+v+1), R:pc+3, d
                    y rz (e1, pc1) -----> rze
 si x= rze "error"
 si no
    s   e   pc   d   ----->   x, s   e   pc+3   d
```

En el caso de que encuentre una receta, se marca como en curso de

evaluación, rze, y se pasa a la pila D la dirección del elemento del vector donde se encuentra. El hecho de guardar en D la palabra V: m+v+1 es necesario para que, una vez evaluada la receta, se sustituya por su valor.

El retorno de la receta encontrará en la pila D una palabra de tipo "V" que le indica que se trata de la dirección donde se encuentra la receta que acaba de evaluar, y que esa dirección es la de un elemento de un vector. De este modo, es necesario añadir una transición a la instrucción RET descrita con anterioridad:

si d1= V: dir ent

----------> s e pc d2

y (dir) = x

y que tiene como efecto, sustituir el elemento cuya dirección viene dada por "dir", por el valor recién calculado de "x".

En relación con las optimizaciones LD0 v y LD00 , ya que son un caso particular de LD n v, simplemente queremos comentar que:

- Para acceder al elemento "0 v" del contexto, la expresión 1(e) nos da directamente el vector buscado, vector(n,m).

- Para acceder al elemento "0 0", la expresión 1(e) nos da directamente el elemento buscado, ya que 1(e) es un "vector(n,m)", y "m+1", la dirección del elemento.

Representamos a continuación, en una tabla, el número de accesos que debe realizar cada una de estas instrucciones, tanto en la representación primitiva, como en la que nos ocupa en este apartado.

Ins.	LD n v	LD0 v	LD00
Repr.			
listas	n+v+2	v+2	2
vectores	n+2	2	2

En el capítulo 5, se muestran los resultados obtenidos, al medir el

número de accesos realizados en la ejecución de varios programas de prueba, utilizando ambos tipos de representación para el contexto, listas y vectores. Se comprueba, que utilizando vectores, el número de accesos que es necesario realizar al contexto a lo largo de la evaluación de los programas se ve reducido en algo más de un 50 por ciento, lo que prueba la eficiencia de la representación propuesta.

Aplicación de Funciones de usuario

17.- AP

```
x, vec(n, m), s   e   pc   d   ----->
si x= apt(el, pcl) ent
        -----> s   (vec(n, m).el)   pcl+1   E:e, R:pc+1, d
si x= número ent
 sea v el primer elemento del vector (que deberá ser el único)
 si x= 0 ent
        ----->   v,s   e   pc+1   d
 si x= -1 ent
        ----->   v,s   e   RCD   R:pc+1, d
 si x= 1 ent
        ----->   v,s   e   RCA   R:pc+1, d
 si no
        ----->   v,x,s   e   RCN   R:pc+1, d
```

Cuando se encuentra un "apunte" en la pila S, se pasa a aplicar la función al vector que contiene los argumentos reales de la función, y que está en la siguiente palabra de la pila S.

Cuando encuentra un número, el vector será unitario. Su elemento será la lista a la que se debe aplicar el selector. Nótese la diferencia con la misma situación vista en el apartado 3.2, utilizando únicamente la representación de listas. En aquel caso, había que deshacer la lista de argumentos formada, para quedarnos con el primero de sus elementos.

18.- VRF

Esta instrucción se pasa a ser una instrucción de una sola palabra, quedando reducida por lo tanto al código de operación, ya que encontrará la pila S de la forma:

x, vec(n,m), s

y por consiguiente, no necesita llevar información acerca del número de argumentos reales, ya que éste le viene dado en el mismo objeto "vector". El número de argumentos reales es el número de elementos que componen el vector. Por ello no es necesario modificar las transiciones que definen esta instrucción, basta con modificar el origen del valor "número de argumentos reales".

Una vez descritas las modificaciones que impone la utilización de vectores para representar la información de contexto, resta únicamente evaluarla para comprobar si realmente es tan eficiente como en principio cabe suponer. Aunque ya hemos adelantado algunos de los resultados, dedicaremos parte del capítulo 5 a tratar este tema en profundidad.

CAPITULO 4

MANEJO DE FICHEROS EN UN SISTEMA FUNCIONAL

4. MANEJO DE FICHEROS EN UN SISTEMA FUNCIONAL

El conjunto de los programas que operan siempre sobre los mismos datos, conocidos en tiempo de compilación, y persiguiendo, por tanto, los mismos resultados, son realmente un subconjunto muy reducido de los que normalmente se necesitan. En general, un programa toma sus datos de algún fichero del sistema, en tiempo de ejecución, y proporciona un resultado que queda a su vez reflejado en algún fichero, asociado a un dispositivo del sistema.

En contra de lo que ocurre en los lenguajes de programación convencionales, en los lenguajes funcionales no existen, de forma explícita, sentencias de lectura o escritura (read o write). Esto es debido a que las definiciones clásicas de estas funciones incluyen la aparición de efectos secundarios (side-effects) en el estado de los ficheros y dispositivos a los que hacen referencia. Estos efectos repercuten en las posteriores llamadas a estas funciones y, por consiguiente, son claramente incompatibles con la propiedad de "transparencia referencial" que caracteriza a los lenguajes aplicativos o funcionales.

Una forma natural de incluir el concepto de fichero en un lenguaje funcional es hacer que éstos sean los argumentos y resultados del programa, como plantean Friedman y Wise [19], y un modelo útil para introducir ficheros en un sistema funcional es el del concepto de "flujo", introducido por Landin [33] y posteriormente utilizado por Burge [6]. Un flujo es un tipo especial de función, sin argumentos, que representa una secuencia y devuelve como resultado un par de elementos, cuyo primer componente es el primer elemento de la secuencia y cuyo segundo componente representa el resto de la secuencia. La aparición de este tipo de funciones surge en principio para representar secuencias infinitas. Por ejemplo, una secuencia infinita de ceros se puede representar, utilizando este concepto, de la forma:

```
ceros ()
donde-rec ceros = función ()  0 : ceros ()
```

Además, el operador ":" de construcción de listas debe evaluar su primer argumento, 0, y dejar el segundo "suspendido". Es decir, en un flujo, el operador de construcción de listas debe ser estricto en su primer argumento y no debe serlo en el segundo.

Hemos desarrollado un sistema de gestión de ficheros para un sistema funcional, basado en la utilización de este tipo de funciones. Se dedica este capítulo a presentar este sistema, que se ha realizado en el VAX 11/750, con sistema operativo VMS, del Centro de Cálculo de la Facultad de Informática de Madrid. La primera parte del capítulo se dedica al manejo de varios ficheros de entrada. En la segunda parte, nos ocuparemos del manejo de varios ficheros de salida. En ella proponemos un modelo basado en la creación de varios procesos, cada uno asociado a un fichero de salida y encargado de realizar las evaluaciones necesarias para obtener el resultado correspondiente. De este modo conseguimos gestionar, de modo asíncrono, cada uno de los ficheros de salida.

4.1.-FICHEROS DE ENTRADA

La idea de utilizar el concepto de flujo para el tratamiento de los ficheros de entrada a un programa funcional aparece ya materializada en Mañas[34], donde un fichero resulta ser una lista unidimensional de números y símbolos, y donde se introduce la función primitiva "fich()" que representa un fichero de entrada, que en ése caso era único, y que el usuario debía especificar en tiempo de ejecución. Se apuntaba, sin profundizar en ello, que esta función podría tener un argumento, cuya evaluación proporcionase información acerca del fichero de que se tratase.

Hemos materializado la idea de introducir un argumento en la función "fich", para conseguir que un programa pueda trabajar, en general, sobre cualquier fichero o ficheros de entrada. De este modo, aparecen en el programa expresiones del tipo:

```
x = fich (id)
```

donde se indica que "x" es un fichero de entrada, cuyo nombre se obtiene al evaluar el argumento "id". Para nosostros, un fichero es una secuencia de

caracteres, concepto similar al utilizado por UNIX y, por lo tanto, no establecemos diferencia entre números y símbolos, dejando al usuario la labor de agrupar los caracteres que contiene el fichero, en átomos o listas, pudiendo facilmente formar un símbolo ya que, en realidad, un símbolo no es más que una secuencia de caracteres.

El argumento "id", puede ser, una simple cadena de caracteres que representa el nombre del fichero, como por ejemplo sucede en la expresión:

```
x = fich ([cuenta.subc]datos.dat)
```

que corresponde a la sintáxis utilizada por VMS para referenciar un fichero, o bien una expresión más compleja, como por ejemplo:

```
x = fich (fich (tt:))
```

utilizada para pedir al usuario el nombre del fichero de entrada desde el terminal, durante la ejecución del programa.

Es importante, destacar la diferencia existente entre la evaluación de la expresión:

```
x = fich (id)
```

y la ejecución de su análoga en los lenguajes imperatívos, "open (id)". Mientras que la primera nos aporta virtualmente en "x" todo el fichero (para ser más exactos, una lista formada por su primer caracter y una "promesa de fichero" que representa el resto del fichero), la segunda proporciona simplemente la posibilidad de accederle, mediante posteriores instrucciones de lectura, "read (id)".

Esta es una forma sencilla de incluir la especificación de un fichero en cualquier punto del programa donde sea necesario. Consideramos esta alternatíva mucho más adecuada que la de Friedman [19], que restringe el uso de los ficheros a la cabecera del programa, obligando a que los argumentos de la función principal sean siempre ficheros, hecho que, por otra parte, no tiene por qué ser siempre cierto.

4.2.-FICHEROS DE SALIDA

En nuestro sistema, la especificación de los ficheros de salida se mantiene exterior al lenguaje de programación, al igual que ocurre en el de Friedman [19]. El usuario debe especificar el fichero donde desea el resultado, en el momento de comenzar la evaluación de su programa.

Al ser el nuestro, un sistema basado en el concepto de "evaluación retardada" (Henderson [26]), no se evalúa nada de forma inmediata, sino que se construyen las llamadas "suspensiones" o "recetas", que sólo se evalúan cuando alguien necesita su resultado o, dicho de otro modo, cuando su evaluación es necesaria para la obtención del resultado final del programa. Así pues, el resultado mismo del programa es, en principio, una estructura cuya evaluación está "suspendida".

Ya que nuestro deseo es enviar el resultado de la evaluación del programa a un dispositivo secuencial, su controlador deberá recorrer la estructura que representa el resultado. Este recorrido se realiza mediante las funciones CAR y CDR, que fuerzan la evaluación de las recetas aún pendientes. Por tanto, sólo se realizarán los cálculos necesarios para averiguar cúal es el siguiente caracter a escribir, convirtiéndose el controlador en el inspirador de todas las evaluaciones que se realizan. Estamos pues en un sistema, donde la evaluación viene dictada por la salida de resultados ("output-driven").

Al principio de este apartado, hablamos de un solo fichero de salida. En realidad, nuestro sistema incluye la posibilidad de especificar varios ficheros de salida. De este modo, el usuario especifica, al arrancar la evaluación de su programa, los ficheros donde desea depositar sus resultados, siendo un caso particular aquel en el que tenemos un solo fichero de salida.

Aunque, en principio no es obvio el hecho de que un programa funcional, basado en la en la utilización y aplicación de funciones, devuelva mas de un resultado (del mismo modo que no es obvio concebir una función con más de un resultado), es a veces no solo útil sino necesario que un programa proporcione varios resultados, cada uno de ellos en un fichero,

para utilizarlos después, por ejemplo, como datos de entrada a otros programas. La solución que proponemos, y que hemos adoptado en nuestro sistema, se basa en la solución clásica de utilizar una estructura tipo "registro", que contenga los diversos resultados de una función. Por ejemplo, en un lenguaje clásico como Pascal, se puede escribir un programa que utilice una función que proporcione dos resultados: la suma y el producto de sus argumentos, de la forma:

```
type resultado: record
                suma: integer;
                prod: integer;
                end
var h: resultado;

function suprod (x: integer; y: integer): resultado;
   var m: resultado;
   begin
      m.suma:= x+y;
      m.prod:= x*y;
      suprod:= m
   end

begin
   h:= suprod (4, 2);
   write (h.suma);              (*  primer resultado  *)
   write (h.prod)               (*  segundo resultado  *)
end.
```

En nuestro caso, el programador no tiene que preocuparse de descomponer el resultado y volverlo a componer, como sucede en el ejemplo escrito en Pascal más arriba. La idea consiste en desglosar el resultado de forma dinámica y no, programar un resultado compuesto de forma estática. Esta es una de las diferencias fundamentales con respecto al concepto de "combinación funcional" de Friedman [21] que además, hace que no sea necesario incluir nuevas y complejas estructuras en el lenguaje como ocurría allí.

El resultado de un programa funcional es a, nuestro entender, una

lista compuesta por cada uno de los resultados que el usuario desea obtener. Según esto, una de las formas de escribir el programa anterior, en un lenguaje funcional, es la siguiente:

```
suprod (4, 2)
donde suprod = función (x, y)
                        (x+y) : (x*y)
```

El "car" de la lista construida es el primer resultado deseado, y el "cdr" el segundo, esto es, la suma y el producto de los argumentos de "suprod" respectívamente.

Hemos realizado la especificación de los ficheros de salida, en base al conocimiento que tiene el programador de la estructura del resultado múltiple de su programa. La solución consiste en especificar los ficheros de salida como una lista, cuyos elementos son los nombres de estos ficheros, y cuya estructura ha de ser idéntica a la estructura de nivel superior del resultado. En el ejemplo anterior, cuyo resultado es la lista:

```
((x+y) . (x*y))
```

si queremos obtener cada uno de sus componentes en un fichero, por ejemplo en los ficheros "suma.dat" y "prod.dat", basta con especificar estos ficheros de la forma:

```
(suma.dat & prod.dat)
```

La sustitución del símbolo "." por el símbolo "&", es un simple detalle de conveniencia, para no confundir el símbolo "." de construcción de listas, con el caracter "." que puede formar parte del nombre de un fichero.

A la vista de la estructura de los ficheros de salida, el sistema realiza el desglose adecuado del resultado, enviando cada una de las partes obtenidas al fichero correspondiente. Además, cada uno de los ficheros es entregado a un controlador, encargándose éstos últimos de ordenar la evaluación de la parte que les corresponde.

Pasamos, de esta forma, del concepto de programa monotarea al de programa multitarea. A cada tarea se le encarga la evaluación de un resultado y se le asigna un fichero de salida donde lo depositará. Los resultados se pueden evaluar concurrentemente y por lo tanto, las tareas se gestionan de forma concurrente, repartiéndose el tiempo de proceso del sistema, en el caso de un sistema monoprocesador. Existe, sin embargo, una cierta interacción entre las tareas, ya que comparten subexpresiones comunes. El sistema se encarga de gestionar el acceso a estas subexpresiones y descubrir y resolver el problema de la existencia de definiciones circulares en el programa, mediante el paso de un conjunto mínino de mensajes entre tareas.

El acceso a subexpresiones comunes se realiza de forma exclusíva, de tal forma, que las zonas críticas sean lo mas pequeñas posibles, para evitar una excesíva contención entre tareas.

La memoria se gestiona de forma centralizada y está dividida en bloques de tamaño fijo, que se van asignando a las tareas según lo van necesitando. Cada tarea dispondrá de un bloque de memoria para realizar sus evaluaciones, de forma que no exista competencia entre varias tareas por la utilización de una celda de memoria libre. Hemos incluido una tarea especial del sistema, que llamamos Gestionador de Memoria, que es la encargada de conceder bloques de memoria libres a las tareas y ordenar la recolección en caso de que se haya acabado la zona de memoria libre.

Es necesaria, en consecuencia, una cierta interacción entre las tareas que evalúan el programa y el Gestionador de Memoria. Las tareas no pueden "apropiarse" indiscriminadamente de un bloque de Memoria ni iniciar la recolección de su información independientemente del resto del sistema. El sistema proporciona mecanismos seguros de comunicación entre las tareas y el Gestionador, basados igualmente en el paso de mensajes entre ambos tipos de tareas.

En lo que sigue, exponemos el sistema de evaluación obtenido en base a las consideraciones anteriores. Trataremos en detalle los aspectos de generación, sincronización y comunicación entre tareas, así como algunos detalles importantes de realización. Por último trataremos la solución pro-

puesta al problema de la gestión de la Memoria común.

4.2.1.- SISTEMA DE EVALUACION MULTITAREA

Se dedica este apartado a describir el sistema funcional desarrollado, y que constituye la parte fundamental de este trabajo. En él coexisten varias tareas, cada una de ellas encargada de la evaluación de una parte del resultado de un programa funcional.

Es importante destacar, que el mecanismo de evaluación que proponemos es independiente de la forma en la que se gestione la Memoria del sistema. Para poder llevar a cabo la evaluación de un programa, es necesario únicamente disponer de espacio de memoria libre, la forma en que ésta se consiga no afectará en ningún caso a la obtención del resultado. Por ello, no hemos considerado en este apartado ninguno de aspectos relatívos a cómo consiguen las tareas el espacio de memoria que necesitan, aspectos que trataremos en el apartado 4.2.3.

El sistema consta de los siguientes componentes:

- Tareas
- Recetas
- Mensajes

A continuación, pasamos a describir cada uno de ellos.

TAREAS

Las tareas son las encargadas de evaluar los resultados de un programa. Existe en el sistema una tarea por cada resultado que se desea obtener, o lo que es lo mismo, por cada fichero de salida especificado por el usuario. Una tarea queda definida por la siguiente información:

- La identificación o nombre de la tarea: ID
- Su estado de Evaluación: SE_{ID}
- Su pila de resultados S: S_{ID}
- Su pila de estados D: D_{ID}

- Un registro contador de programa que contiene la dirección de la zona de código a ejecutar por la tarea: PC_{ID}
- El contexto donde se debe realizar la evaluación: E_{ID}
- El fichero donde dirigirá el resultado obtenido: FS_{ID}

Una tarea del sistema puede estar en alguno de los estados: LATENTE, ACTIVO, ESPERA o MUERTO.

- LATENTE:

Una tarea está en estado LATENTE, lo que designaremos como "L", cuando el sistema aún no ha arrancado su evaluación, pero en algún momento, durante la ejecución del programa, lo hará con seguridad. Excepcionalmente pueden producirse errores que hagan abortar prematuramente al sistema.

Una tarea "i" en este estado la representaremos de la forma:

$$T_i\ (L)$$

- ACTIVA:

Una tarea está en estado ACTIVA, lo que designaremos como "A", cuando está funcionando normalmente, es decir, cuando está realizando las evaluaciones que le han sido asignadas.

Una tarea en este estado la representaremos de la forma:

$$T_i\ (A,\ D_i,\ QV_i)$$

donde D_i representa la pila de estado D de esta tarea, que contendrá entre otras cosas, las recetas de cuya evaluación se está haciendo cargo y QV_i una cola de mensajes, cuyo significado se verá más adelante en este apartado.

- ESPERA:

Una tarea está en estado de ESPERA, lo que designaremos como "$E(R_m,\ T_j)$", cuando necesita conocer el valor de una expresión, R_m que está evaluando otra tarea T_j, y depende de ella en el sentido

de que debe esperar a que se termine su evaluación para poder continuar.

A una tarea en este estado la representaremos de la forma:

$$T_i\ (E(R_m,T_j),\ D_i,\ QV_i)$$

donde D_i y QV_i tienen el mismo significado visto en el estado anterior.

- MUERTO:

Una tarea está en estado MUERTO, lo que designaremos como "M", cuando ha acabado la evaluación del resultado que le fué asignado y por lo tanto no tiene trabajos pendientes.

A una tarea en este estado la representaremos de la forma:

$$T_i\ (M,\ QV_i)$$

El resto de la información no necesita mayores comentarios, ya que se vió en el capítulo 3. Unicamente queda, a este respecto, exponer de qué forma obtiene una tarea esta información. Trataremos en detalle este tema en el apartado 4.2.1.1.

RECETAS

Las recetas constituyen el objeto a evaluar. Anteriormente definimos una receta como una estructura compuesta por dos tipos de información: una referencia al contexto en la que fué definida, y la dirección donde se encuentra el código a ejecutar para conocer su valor: rz (el, pcl). Allí la definimos de la forma:

```
receta = record
        contexto: dirección_lista;
        código: dirección_código
         end;
```

Para cuando se arranca su evaluación, hecho que identificamos cambiando el tipo rz de este objeto, por el tipo rze, en nuestro caso es necesario conocer:

a) Quien se está haciendo cargo de la evaluación de la receta.

b) Qué tareas están en ESPERA de que termine la evaluación de esta receta para poder continuar.

El hecho de poder conocer quien se está haciendo cargo de la evaluación de una receta, es de especial importancia. Si, en el sistema monotarea, encontrar una receta marcada como en curso de evaluación (rze), significa la existencia de una definición circular no resoluble por la máquina, en el sistema multitarea, significa simplemente que hay que esperar a que termine su evaluación y se reescriba con un valor. Si en el programa no existen definiciones circulares, la evaluación de las recetas termina con un valor. Si existen, la evaluación de las recetas involucradas en el bucle de definiciones, termina con una lista de tipo error. Para detectar estas situaciones erróneas, hemos desarrollado un mecanismo de paso de mensajes entre tareas, que exponemos en el apartado 4.2.1.2.

En definitíva, una receta en curso de evaluación es una estructura de la forma:

```
type rze = record
             i: nom_tarea;      (* nombre de la tarea que
                                   evalúa la receta     *)
             W: lista           (* lista de tareas en ESPERA
           end;                    de que termine su evaluación *)
```

y que representaremos cómo: R_m <T_i, W_m>, siendo "m" la identificación de la receta. Esta identificación corresponde, realmente a la dirección de la celda de memoria en la que se encuentra, y únicamente se ha introducido para aclarar el hecho de que dos tareas encuentren la misma receta.

El acceso a las recetas debe hacerse de forma exclusíva, y la operación de cambiar de rz (e1, pc1) a R <T_i, W>, debe realizarse de forma

interbloqueada. El acceso indiscriminado a las recetas, podría dar lugar a situaciones erróneas dificilmente detectables. Por ejemplo, si dos tareas, llamémoslas T_1 y T_2, intentasen acceder a la misma receta $\underline{rz}_1$(el, pcl), podría darse el caso de que se llegase finalmente a una situación en la que:

- T_1 (A, R_1:D_1, QV_1)
  T_1 se ha hecho cargo de la receta

- T_2 (A, R_1:D_2, QV_2)
  T_2 también se ha hecho cargo de la receta

- R_1 <T_2, vacía>
  La receta está siendo evaluada por T_2 y no hay ninguna tarea esperando por ella.

MENSAJES

Los mensajes son objetos creados para realizar la comunicación entre las distintas tareas del sistema y tienen como finalidad:

- Que una tarea, al terminar la evaluación de una receta, pueda informar de este hecho a las tareas que esperaban por ella, para que estas ultimas puedan continuar.

- Poder detectar y romper bucles de dependencias entre tareas para que el sistema no quede "bloqueado". De esta forma, la ejecución del programa puede continuar, quedando reflejado este hecho mediante la aparición de listas de tipo error, sin que se vean afectadas las tareas que no están en ese bucle.

Estos mensajes, que denominaremos "mensajes de Evaluación, ME", constan de dos tipos de información, el tipo de mensaje de que se trata, y las tareas y recetas involucradas. Son los siguientes:

[I. T_i. R_m]
  Mensaje mediante el cual la tarea T_i informa que espera el

valor de la receta R_m.

[F]

Mensaje mediante el cual una tarea informa a otra que ha finalizado la evaluación de la receta que esperaba.

La existencia de los objetos Mensajes, está soportada por medio de una cola FIFO asociada a cada tarea, y que hemos denominado QV. QV_i es la cola de mensajes de Evaluación de T_i.

En el apartado 4.2.1.2 veremos en detalle cómo, cuando y por qué aparecen estos mensajes, así como la función que desempeñan en el sistema.

Hasta aquí, hemos definido los componentes básicos del sistema de evaluación. En lo que sigue, describiremos el mecanismo de generación de tareas y su relación con la estructura de los ficheros de salida, así como el modelo experimental de tareas desarrollado.

4.2.1.1.- GENERACION DE TAREAS

El aspecto de generación de tareas está ligado directamente a la especificación de los ficheros de salida, ya que la aparición de tareas en el sistema es, en nuestro caso, consecuencia directa del número de ficheros de salida que especifica el usuario. Para ser mas exactos, es consecuencia directa de la estructura de ficheros de salida, que debe ser isomorfa a la estructura de nivel superior del resultado del programa funcional. Por ejemplo, la lista de ficheros (f1 & f2) corresponderá a un resultado de la forma (a.b), donde "a" y "b" pueden ser a su vez listas.

En el caso del programa monotarea, como vimos en el capítulo anterior, se arrancaba la evaluación del programa con la rutina de impresión, compuesta por las instrucciones IMP y RET, en la pila de estado D. Esta rutina era la encargada de arrancar la evaluación del resultado. Al introducir varias tareas en el sistema, hay que partir de algo más elaborado, que permita al sistema identificar cuando se debe arrancar una nueva tarea, para que ésta a su vez inicie la evaluación de un resultado. Este algo más elaborado, dependerá por tanto de la estructura de ficheros de sa-

lida. A cada fichero de salida se le asigna una tarea, de la forma que se detalla a continuación.

- Primero, se recorre la lista de ficheros de salida, ignorando su estructura, para asociar un número a cada uno de los ficheros. Esto equivale a "aplanar" la estructura de ficheros de salida y formar a partir del resultado una lista cuyos elementos son a su vez listas compuestas por el nombre de un fichero y un número asociado a él. La obtención de esta lista la describimos en función de "asigna", función que describimos a continuación.

```
sea  asigna (x, l)      -- resultado, empezando la asignación
                        -- a partir del número l

  y  asigna= función (xl, n)
                 si xl = NIL ent  xl : n
                 si no ((l  x) : n) : asigna ((-l x), n+l)

  y  x= aplana (lf)     -- aplanar la lista de ficheros de
                        -- salida, denominada lf

  y  aplana= función (lis)
                 cond
                 si lis = NIL ent NIL
                 si ATM (l lis) ent (l lis) : aplana (-l lis)
                 si no append (aplana (l x), aplana (-l x))

  y  append= función (x, y)
                 cond
                 si x = NIL ent
                              si y = NIL ent NIL
                              si no y
                 si y = NIL ent x
                 si no (l x) : append ((-l x), y)
```

- Se emplea la lista obtenida en el paso anterior para generar el código de la rutina de salida. Describiremos este segundo paso en función de "genera", función que devuelve el código generado a partir de una lista de ficheros, y la lista anterior:

```
sea genera (lf, tabla)

   genera = función (e, t)
       si ATM (e) ent LDC # e # OPEN # IMP # RET
       si no FORK # no((-1 e), t) # CDR # genera(-1 e)
                # CAR # genera(1 e)

y no = función (x, t)                    -- devuelve el número
                                         -- asociado a un fichero
       si ATM (x) ent  busca(x, t)
       si no  no((1 x), t)

y busca = función (nom, t)    -- busca en "t" el número asociado
                              -- al fichero de nombre "nom"
          cond
          si nom = 1 (1 t) ent -1 (1 t)
          si no busca (nom, (-1 t))

y tabla = asigna (aplana (lf), 1)    -- lista obtenida en el paso
                                     -- anterior
```

En esta definición del proceso, hemos utilizado la notación "#" para indicar una estructura lineal. "a # b", significa que "a" y "b" se almacenan en posiciones contiguas de memoria, sin que exista ningún apuntador de "a" a "b".

Aclararemos la generación de código con un ejemplo sencillo. Sea la lista de ficheros de salida (a & b), correspondiente al resultado de un programa de la forma (r1.r2). En el primer paso descrito obtendríamos la lista de asignaciones:

```
((a.1) (b.2))
```

y en el segundo paso, obtendríamos el código:

```
FORK 2 T2          T2: CDR
CAR                    LDC b
LDC a                  OPEN
OPEN                   IMP
IMP                    RET
RET
```

Cuando la tarea que inicia la ejecución del programa ejecuta la instrucción FORK 2 T2, crea una tarea de nombre "2" y le pasa el código etiquetado con T2 para que lo ejecute. Este código corresponde a la evaluación del resultado "r2". Ella misma se encarga de ejecutar el código que aparece a continuación, que no es otra cosa que el código necesario para evaluar el resultado "r1".

Con una estructura más compleja de ficheros de salida, por ejemplo:

((a & b) (c & d) & e)

obtendremos, asignando a los ficheros a, b, c , d y e, los números 1, 2, 3, 4 y 5 respectivamente, el código:

```
FORK 3 T3        T2: CDR          T3: CDR
CAR                  LDC b            FORK 5 T5
FORK 2 T2            OPEN             CAR
CAR                  IMP              FORK 4 T4
LDC a                RET              CAR
OPEN                                  LDC c
IMP                                   OPEN
RET                                   IMP
                                      RET
```

```
T4: CDR          T5: CDR
    LDC d            LDC e
    OPEN             OPEN
    IMP              IMP
    RET              RET
```

Intrucción FORK n pc1

Arranca una nueva tarea a la que identifica con el nombre "n", y que será la encargada de ejecutar el código de la zona de memoria a la que apunta "pc1".

La instrucción queda definida por el siguiente algoritmo:

1.- [INICIALIZAR NUEVA TAREA]
sea "i" el nombre de la tarea que ejecuta la instrucción FORK.

ID = n	Nombre por el que se conocerá a la nueva tarea en el sistema, "n"
SE_n = A	Estado, ACTIVA, para que pueda iniciar inmediatamente su ejecución
S_n = S_i	Pila S igual a la de la tarea "i" información acerca del resultado que debe evaluar.
D_n = vacía	Pila D vacía No tiene ningún estado pendiente.
PC_n = pc1	Código a ejecutar, "pc1" rutina de impresión de esta tarea
E_n = E_i	Contexto en el que debe evaluar el resultado, igual al de la tarea "i"

2.- [ARRANCAR NUEVA TAREA]
Incorporar a la nueva tarea al conjunto de tareas existentes en el sistema.

Instrucción OPEN

Esta instrucción es la encargada de asignar a una tarea el fichero donde deberá dejar su resultado. Encontrará, en la cima de la pila S, el nombre del fichero de salida que le fué asignado, y queda definida por la transición:

id, s e pc d ---------> s e pc+1 d

donde "id" es el nombre del fichero. Tiene como efecto la obtención del valor de FS de la tarea que la ejecuta, FS_n = id.

A la vista del código generado, se puede observar cómo se genera un árbol binario de tareas, dependiendo su profundidad del número de anidamientos de la estructura de ficheros de salida. No existe, sin embargo, una dependencia especial entre las tareas de niveles más altos y las de niveles más bajos. En particular, no existe dependencia especial entre la tarea que lanza a otra y la tarea lanzada. Esta última puede continuar, independientemente de que la otra pase o no al estado de MUERTO. La única dependencia existente entre tareas, se debe a la compartición de estructuras comúnes durante la evaluación de un programa, y ésta es independiente del nivel del árbol en que esté la tarea.

4.2.1.2.- MODELO EXPERIMENTAL DE TAREAS

En este apartado definimos formalmente, el sistema funcional de evaluacion presentado en el apartado anterior.

Definimos el sistema funcional S_F, como una tupla:

$$S_F = < T, R, ME >$$

donde T es un conjunto de tareas, que actúan sobre un conjunto de recetas R y se coordinan por medio de un conjunto de mensajes ME.

Una TAREA queda definida por la siguiente información:

- Su nombre
- Su estado de Evaluación
- Su pila D
- Su cola de mensajes de evaluación

La representaremos de la forma:

$$T_i\ (SE_i,\ D_i,\ QV_i)$$

Las situaciones en las que se puede encontrar una tarea son las correspondientes a su estado. Las representaremos mediante la notación introducida en el apartado 4.2.1.

$T_i\ (L)$	Tarea en estado LATENTE
$T_i\ (A,\ D_i,\ QV_i)$	Tarea ACTIVA
$T_i\ (E\ (R_m,\ T_j),\ D_i,\ QV_i)$	Tarea en ESPERA
$T_i\ (M,\ QV_i)$	Tarea en estado MUERTO

En los casos primero y último, no aparece la información D_i ya que una tarea que está en estado L o M, no tiene evaluaciones pendientes. En el primer caso, aún no ha arrancado, y en el último, ya ha finalizado sus evaluaciones. La cola de mensajes de Evaluación tampoco aparece, en el caso de una tarea en estado L ya que está vacía. Una tarea solo puede recibir un mensaje de Evaluación cuando está evaluando alguna receta y por lo tanto está ACTIVA, cuando está en ESPERA de que otra tarea termine la evaluación de la receta que necesita para poder continuar, o cuando está en estado MUERTO y existen tareas que aún no están enteradas de que ha finalizado sus evaluaciones.

Las RECETAS quedan definidas por:

- su nombre
- su estado
- información dependiente del estado.

Una receta, m, puede estar en alguno de los estados:

- sin evaluar, lo denominaremos como RZ
- en evaluación, lo denominaremos como R

Para desarrollar el modelo, únicamente nos interesa la información de una receta cuando está en curso de evaluacion, ya que este hecho supone una comunicación entre la tarea que la evalúa y las que necesitan su valor. En el caso de una receta sin evaluar, no es necesaria comunicación alguna entre tareas, basta con asegurar un acceso exclusivo a la receta, de forma que sea una sola tarea la que se haga cargo de su evaluación.

Una receta en evaluación, caracterizada por el estado R viene definida finalmente por:

- su nombre: i
- la tarea que se encarga de su evaluación: T_j
- la cola de tareas que esperan su resultado: W_i

representamos de la forma:

$$R_i \langle T_j, W_i \rangle$$

Los MENSAJES de Evaluación, mediante los cuales se realiza la coordinación entre tareas los representaremos mediante la misma notación utilizada para su descripción:

$$[\ I.\ T_i.\ R_m\ ] \in QV_j$$
$$[\ F\ ] \in QV_j$$

indicando de este modo que el mensaje está en la cola de mensajes de una cierta tarea, T_j.

Un programa funcional lo representaremos en términos de tareas y recetas. A esta representación la denominaremos <u>historia</u> del programa. Sea, por ejemplo, el programa funcional del que se quieren obtener dos resultados x e y:

```
sea x : y
  x= 1+y
  y= 2*z
  z= 4
```

Lo representaremos, mediante su "historia" de la forma:

$T_1 = R_1$	T_1 evalúa la receta R_1 correspondiente al valor de x
$T_2 = R_2$	T_2 evalúa la receta R_2 correspondiente al valor de y
$R_1 = R_2$	Para evaluar R_1 (x), es preciso conocer el valor de R_2 (y)
$R_2 = R_3$	Para evaluar R_2 (y) es preciso conocer el valor de R_3 (z)

Una vez definidos los objetos y la notación utilizadas, pasamos a definir las reglas que establecen el comportamiento del sistema.

I.- Reglas que definen la actuación de las tareas

RT-1

Inicialmente:

Todas las tareas del sistema estan en estado L: T_i (L) $\forall T_i \in \{T\}$

Al crearse las recetas, están sin evaluar, y no existen por lo tanto colas de tareas en Espera.

RT-2

Una tarea T_i se Actíva mediante la ejecución de una instrucción FORK i pcl.

$$T_i(L) \text{ -----> } T_i(A, \text{ vacía, vacía})$$

RT-3

Una tarea activa T_i, sin mensajes, puede comenzar a evaluar una receta RZ_m.

$$T_i(A, D_i, \text{vacía}) \text{ -----> } T_i(A, R_m{:}D_i, \text{vacía})$$
$$\underline{y} \quad RZ_m \text{ -----> } R_m\langle T_i, \text{vacía}\rangle$$

RT-4

Una tarea activa T_i, sin mensajes, puede desear conocer el valor de una receta R_m, de cuya evaluación se está haciendo cargo otra tarea T_j. Informa de este hecho a T_j y queda a la Espera de que acabe su evaluación.

$$T_i(A, D_i, \text{vacía}) \text{ -----> } T_i(E\ (R_m, T_j), D_i, \text{vacía})$$
$$\underline{y} \quad QV_j \text{ -----> } QV_j : [\ I \cdot T_i \cdot R_m]$$
$$\underline{y} \quad R_m\langle T_j, W_m\rangle \text{ -----> } R_m\langle T_j, W_m{:}T_i\rangle$$

RT-5

Una tarea activa T_i puede terminar la evaluación de la última receta de la que se hizo cargo. La saca entonces de su pila D, reescribiéndola con su valor e informa de este hecho a las tareas que estén esperando por ella.

$$T_i(A, R_m{:}D_i, \text{vacía}) \text{ -----> } T_i(A, D_i, \text{vacía})$$
$$\underline{y}\ \nexists T_j \in W_m \quad QV_j \text{ -----> } QV_j{:}[\ F\ ]$$
$$\underline{y} \quad W_m \text{ -----> vacía}$$

RT-6

Una tarea activa, sin mensajes ni tareas pendientes puede morir.

$$T_i(A, \text{vacía}, \text{vacía}) \text{ -----> } T_i(M, \text{vacía})$$

II.- Reglas referentes a la recepción de mensajes

RM-1

Inicialmente no existen mensajes en el Sistema.

$$QV_i = \text{vacía} \quad \forall\, T_i \in \{ T \}$$

RM-2

Una tarea T_i en alguno de los estados A, E o M puede recibir un mensaje de tipo I, referente a una receta que no está evaluando, ignorándolo.

$$T_i(SE_i, D_i, [I.\ T_j.\ R_m]:QV_i) \quad \underline{y} \quad R_m \notin D_i \quad \text{-------->} \quad T_i(SE_i, D_i, QV_i)$$

RM-3

Una tarea T_i, en Espera, puede recibir un mensaje de tipo I que le afecta.

$$T_i(E\ (R_n, T_j), D_i, [I.\ T_h.\ R_m]:QV_i) \quad \underline{y} \quad R_m \in D_i$$

$$\underline{si}\ i \diamond h \quad \text{-----> } T_i(E\ (R_n, T_j), D_i, QV_i) \quad \underline{y}\ QV_j \text{ -----> } QV_j : [I.\ T_h.\ R_n]$$

$$\underline{si\ no} \text{ -----> } T_i(E\ (R_n, T_j), D_i, QV_i) \quad \underline{y}\ R_m \text{ -----> "error"} \quad \underline{y}\ \forall T_g \in W_m \quad QV_g \text{ ----> } QV_g : [F] \quad \underline{y}\ W_m \text{ ----> vacía}$$

RM-4

Una tarea T_i Activa, puede recibir un mensaje de tipo I que le afecta.

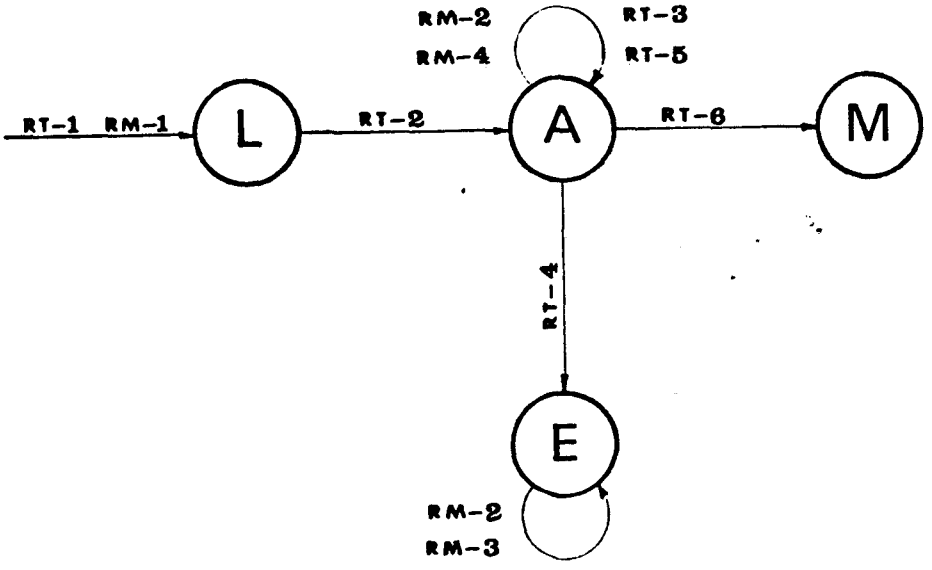
$$T_i(A, D_i, [I.\ T_h.\ R_m]:QV_i) \quad \underline{y} \quad R_m \in D_i \quad \text{-----> } T_i(A, D_i, QV_i)$$

RM-5

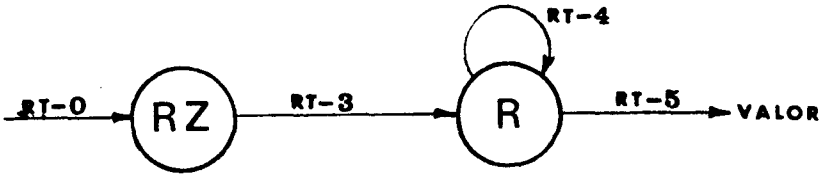
Una tarea T_i en Espera, puede recibir un mensaje de tipo F.

$$T_i(E\ (R_n,\ T_j),\ D_i,\ [F]:QV_i) \longrightarrow T_i(A,\ D_i,\ QV_i)$$

En base a estas definiciones, los grafos de transición de estados de las tareas y las recetas, son los que se muestran a continuación.



- Grafo de transición de las tareas -



- Grafo de transición de las recetas -

III.- Propiedades del Sistema

El modelo de evaluación descrito, tiene las siguientes propiedades:

P1.- El conjunto de reglas que definen su comportamiento es completo.
P2.- Es un sistema que se autodesbloquea, en el caso de que varias tareas formen un bucle cerrado de dependencias.
P3.- Las tareas terminan en un tiempo finito, si los resultados son finitos.

P1.-Completitud del Sistema

El conjunto de reglas que definen el comportamiento del Sistema es completo. Se tienen en cuenta en ellas, todas las posibles situaciones en las que se puede encontrar el Sistema.

El único caso que no se contempla en las reglas y que, por lo tanto, no se puede dar es el hecho de que una tarea Activa reciba un mensaje [F].

Para demostrar que no es posible este hecho establecemos previamente los siguientes lemas.

Lema LC-1.-

Si una tarea T_i está en una cola, entonces está en estado de Espera.

$$T_i \in W_m \quad ====> \quad SE_i = E\ (R_m,\ T_j)$$

demostración

Una tarea solo puede entrar en una cola aplicando la regla RT-4 y ésta hace que pase de Activa a Espera.

Lema LC-2.-

Si una tarea T_i está en una cola, no puede estar en ninguna otra.

$$T_i \in W_m \quad \underline{y} \quad T_i \in W_n \quad ====> \quad m = n$$

demostración

Inicialmente no existen colas, ya que éstas aparecen al comenzar la evaluación de las recetas.

T_i sólo puede entrar en una cola aplicando la regla RT-4. Mientras T_i esté en una cola, sigue en Espera según demostramos en LC-1. Mientras esté en este estado, no se puede aplicar RT-4, por lo que T_i no puede entrar en ninguna otra cola.

Teorema TC-1.-

Si existe un mensaje [F] para una tarea T_i, entonces T_i está en estado de Espera.

$$[\ F\ ] \in QV_i \quad ====> \quad SE_i = E\ (R_m,\ T_j)$$

demostración

Por la regla RT-5, la única forma de que exista un mensaje [F] que pertenezca a QV_i, es que T_i estuviera en una cola, donde la encontró la tarea que le envía el mensaje. Por LC-1, si estaba en una cola entonces estaba en Espera y por LC-2, esta cola es única.

Mientras no se reciba el mensaje [F], T_i seguirá en Espera, aún sin estar en ninguna cola. Solo el hecho de recibirlo le hará pasar a Activa.

Teorema TC-2.-

Si existe un mensaje [F] en la cola de mensajes de una tarea T_i, entonces T_i no puede estar en ninguna cola.

$$\exists\ [\ F\ ] \in QV_i \ ====> \nexists\ W_m:\quad T_i \in W_m$$

demostración

Por el teorema TC-1, si existe un mensaje [F] dirigido a T_i, T_i está en Espera.

Si T_i está en Espera, sólo puede estar en una cola. Pero, [F] aparece en el momento en que T_i se saca de la única cola en la que estaba, aplicando la regla RT-5.

P2.-Autodesbloqueo del Sistema

Puede ocurrir, que varias tareas formen un bucle cerrado de dependencias. Por ejemplo, consideremos la siguiente historia de un programa funcional de dos resultados:

$T_1 = R_1$	T_1 evalúa la receta R_1
$T_2 = R_2$	T_2 evalúa la receta R_2
$R_1 = R_2$	Para evaluar R_1 es necesario conocer el valor de R_2
$R_2 = R_1$	Para evaluar R_1 es necesario conocer el valor de R_1

Este programa tiene un error, que consiste en la aparición de definiciones circulares no resolubles. Para conocer R_1 hay que conocer previamente R_2, pero para conocer R_2 es necesario el valor de R_1. En resumen, para evaluar R_1 es preciso conocer su valor con anterioridad, lo cual es imposible.

En algún instante, durante la ejecución del programa, las dos tareas del sistema estarán en Espera la una de la otra, es decir:

$$T_1(E\ (R_2,\ T_2),\ R_1{:}D_1,\ QV_1)$$

$$T_2(E\ (R_1,\ T_1),\ R_2{:}D_2,\ QV_2)$$

En este instante, decimos que el sistema está "bloqueado".

El conjunto de reglas que definen el comportamiento del sistema, aseguran, que las tareas se autodesbloquean por sí mismas. Para demostrar este hecho, definiremos previamente los conceptos de sucesor inmediato y sucesor de una tarea T_i.

sucesor inmediato

Decimos que T_i es sucesor inmediato de T_j, lo que representaremos de la forma $T_i \rightarrow T_j$, si T_i está en la cola de alguna de las recetas de la pila D_j.

sucesor

Decimos que T_i es sucesor de T_j, lo que representaremos de la forma $T_i \dashrightarrow T_j$, si T_i es sucesor inmediato de T_j o lo es de alguno de sus sucesores. Formalizando esta definición:

$$T_i \dashrightarrow T_j \Longleftrightarrow T_i \rightarrow T_j \ \underline{o}\ \exists T_k: T_i \rightarrow T_k\ \underline{y}\ T_k \dashrightarrow T_j$$

Lema LA-1.-

Si T_i es sucesor de T_j, entonces existe un número finito de tareas que llevan de T_j a T_i.

demostración

El número de tareas en el sistema es finito, ya que lo es el número de ficheros de salida.

Puesto que una tarea solo puede estar en una cola, como demostramos en LC-2, existe una sola tarea T_k tal que:

$$T_i \dashrightarrow T_k \dashrightarrow T_j$$

Por inducción, la cadena que lleva de T_j a T_i sólo puede ser infi-

nita, si existe un T_h que se repite, es decir, si tenemos:

$$T_i \dashrightarrow T_h \dashrightarrow T_p \rightarrow T_h \dashrightarrow T_m \rightarrow \ldots\ldots$$

en cuyo caso, o T_j ya se ha alcanzado antes de llegar a T_p, y el lema es cierto, o nunca llegaremos a T_j, y la hipótesis sería falsa.

Lema LA-2.-

Si una tarea T_i pasa a ser sucesora de otra T_j, entonces en algún momento, existirá un mensaje [I. T_i. R_p] para T_j, o bien T_i dejará de ser sucesor de T_j.

$$T_i(A, D_i, QV_i) \ \underline{y} \ o\, T_i \dashrightarrow T_j$$
$$\Longrightarrow \forall(\exists\, [I.\ T_i.\ R_p] \in QV_j \ \underline{o}\ T_i \not\rightarrow T_j)$$

demostración

Si T_i Activa entra a ser sucesora de T_j, T_i queda entonces a la Espera y envía un mensaje [I. T_i. R_p] a la tarea T_k, de la que ha pasado a ser sucesor inmediato, y que pertenece a la cadena de tareas que llevan de T_j a T_i.

$$T_i \rightarrow T_k \dashrightarrow T_j \ \underline{y}\ \exists [\ I.\ T_i.\ R_p] \in QV_k$$

Al recibir T_k este mensaje pueden ocurrir dos cosas:

a) Que el mensaje le afecte, es decir, que $R_p \in D_k$, en cuyo caso, por la regla RM-3, o bien lo propaga ($i \neq k$), o bien se rompe la cadena (caso de que $i = k$).

b) Que el mensaje no le afecte, es decir, que $R_p \notin D_k$, en cuyo caso significaría que tal receta ya se ha evaluado y, por lo tanto, $T_i \not\rightarrow T_j$

Aplicando este mismo razonamiento iteratívamente a los componentes de la cadena que va de T_j a T_i, y ya que, por el lema LA-1, esta cadena es finita, el mensaje llegará a T_j en un tiempo finito o bien se romperá la cadena.

Teorema TA-1.-

Si una tarea T_i pasa a ser sucesora de sí misma, lo que hemos definido como estado de "bloqueo", en un tiempo finito se romperá la cadena que lleva de T_i a sí misma.

$$T_i(A, D_i, QV_i) \ \underline{y} \ o\,(T_i \rightarrow T_h \ \underline{y} \ T_h \dashrightarrow T_i) \Longrightarrow \nabla T_h \not\rightarrow T_i$$

demostración

Aplicando la regla RT-4, al pasar T_i a ser sucesor inmediato de T_h, le envía un mensaje [I. T_i. R_p].

Al ser T_h sucesor de T_i, y como demostramos en el lema LA-2, el mensaje o bien llegará a T_i, en cuyo caso $T_h \not\rightarrow T_i$, por la regla RM-3, o no llegará debido a que la cadena ya se ha roto antes y por tanto $T_h \not\rightarrow T_i$

Este teorema garantiza que si en el sistema se producen bucles cerrados de tareas en espera, estas se activan solas, produciéndose de este modo el autodesbloqueo del sistema. En particular, para el caso más simple de bloqueo, cuando $T_i \rightarrow T_i$, el teorema asegura que:

$$T_i(A, D_i, QV_i) \ \underline{y} \ o\,T_i \rightarrow T_i \Longrightarrow \nabla T_i(A, D_i, QV_i)$$

P3.- Terminación de las Tareas

Si el resultado o resultados de un programa son finitos, el programa termina en un tiempo finito.

Lema LT-1.-

Si una receta R_m se define recursivamente, en algún momento acaba su evaluación.

demostración

El teorema TA-1 garantiza que acaba su evaluación, ya que la tarea involucrada en su evaluación se autodesbloquea, reescribiendo la receta con un "error".

Teorema TT-1.-

Si dos tareas T_i y T_j necesitan conocer el valor de una receta R_m, no recursíva, para avanzar en su cálculo, entonces eventualmente ambas disponen del valor de R_m.

demostración

Si T_i empieza y termina la evaluación de R_m antes de que T_j llegue, entonces, T_j la encontrará ya evaluada.

Si T_i empieza y T_j la encuentra en evaluación, aplicando la regla RT-4, T_j se pone en Espera hasta que T_i acabe. Aplicando entonces RM-5, T_j se Activa de nuevo, disponiendo ambas tareas del valor de R_m.

Teorema TT-2.-

Si una tarea T_i necesita conocer el valor de la receta R_m, no recursíva, para avanzar en su cálculo y no existe ninguna otra tarea interesada en esa receta, entonces T_i termina de evaluar R_m.

demostración

Si para evaluar R_m no es necesario conocer el valor de ninguna otra receta, es decir, no existe ninguna línea en la historia del programa del tipo $R_m = R_n$, aplicando RT-3 empieza la evaluación de R_m, y aplicando RT-5 acaba.

Si para evaluar R_m es necesario conocer el valor de otras recetas, es decir, existe una linea de la historia del programa del tipo $R_m = R_n$, la aplicación del lema LT-1 y los teoremas TT-1 y TT-2 de forma recursíva es finita, acabando por tanto la evaluación de R_m en un tiempo finito.

Ya que la evaluación de las recetas siempre acaba, y el número de recetas del sistema es finito, en algún momento todas las recetas estarán evaluadas y, por lo tanto las tareas acabarán:

$$\nexists\, R_m \in \{ R \} \Longrightarrow \forall\, T_i \in \{ T \}: T_i(M)$$

4.2.1.3.- DETALLES DE REALIZACION

Debido a que el acceso a las recetas debe realizarse de forma exclusíva, una tarea T_i que accede a una receta para conocer su valor:

- adquiere el acceso exclusívo a la receta
- si RZ_m ent
  - RZ_m ----> R_m <T_i, vacía>
  - libera el acceso exclusívo a la receta
  - continúa su ejecución.

- si R_m <T_j, W_m> ent
  - W_m ----> W_m:T_i
  - libera el acceso.exclusívo
  - envía [I. T_i. R_m] a T_j
  - queda en estado de Espera

En el caso de que la receta esté sin evaluar, la operación dentro de la zona crítica es sencilla, consiste en cambiar el objeto "receta sin evaluar" por el de "receta en evaluación". No lo es tanto la operación que se realiza cuando la receta está siendo evaluada por otra tarea, ya que T_i debe, primero, averiguar de qué tarea se trata y, después ponerse en la cola de la receta.

Para que la zona crítica sea mas pequeña y, por lo tanto, otra tarea que esté esperando entrar en ella espere menos tiempo, hemos hecho que T_i tenga que averiguar, simplemente, quien es la tarea que evalúa R_m y dejar que sea T_j quien la apunte en la cola. Cuando T_i encuentra R_m<T_j, W_m>, libera el acceso a la receta y "dice" a T_j que la ponga en la cola W_m, enviándole un mensaje:

[Q. T_i. R_m] Poner a T_i en la cola de R_m

- si R_m <T_j, W_m> ent
  - libera el acceso exclusívo
  - envía [Q. T_i. R_m] a T_j
  - queda en estado de Espera

Cuando T_j reciba este mensaje, si aún tiene esa receta, realizará la acción:

$$W_m \text{ ----> } W_m:T_i$$

y si ya no la tiene, porque ha terminado de evaluarla, enviará a T_i el mensaje [F], para informarle de que ya puede disponer del valor de la receta que esperaba.

Este nuevo tipo de mensaje aparece como una cuestión importante de realización en cuanto a que reduce la contención entre tareas.

4.2.1.4.- ERRORES EN EJECUCION

En el caso de un sistema monotarea, como el expuesto en el capítulo anterior, el tratamiento de errores, durante la ejecución de un programa, es relatívamente simple, y está basado en las listas de tipo error de Mañas [35]. Se realiza, como vimos entonces, generando un número determinado de listas de éste tipo, y llegado este punto el programa termina, con un resultado que informa al usuario de la expresión de su programa que no se ha podido reducir y el entorno en que ha ocurrido este hecho.

No existe, sin embargo, ningún antecedente de cómo realizar el tratamiento de errores en el caso de un sistema multitarea. En este caso, el problema que aparece cuando una de las tareas detecta un error no puede solucionarse de la misma forma que en el caso anterior. Una tarea no puede acabar sin más, cuando llega a generar un número determinado de listas de tipo error, ya que podría tener recetas en evaluación en su pila de estado D. El hecho de acabar sin más, supondría que las tareas que se encontrasen esperando conocer el valor de estas recetas, quedarían para siempre en espera de recetas que, "supuestamente", está evaluando una tarea Muerta.

Por otra parte, para reescribir estas recetas sería necesario que la tarea acabase de ejecutar el código que aún le queda, y esto significaría seguir propagando el error, cosa que no es posible, pues ya se ha llegado al límite fijado de listas de error.

Proponemos una solución a este problema, que consiste en generar, lo que hemos dado en llamar "error no expandible". Este error está formado por el número máximo de listas de error permitido y, como su propio nombre indica, no se expande a expresiones más externas. Cuando una tarea llega a generar este error, continúa su ejecución sin generar ninguna lista más de error, con el fin de reescribir las recetas que aún tiene en evaluación, con la información del error detectado, y sacar del estado de Espera a las tareas que estén esperando por ellas.

Es importante destacar, que esta forma de resolver las situaciones de error, permite que una tarea acabe de igual forma que si no se hubiera detectado error alguno, y el error no afecte a ninguna de las tareas no involucradas. La única diferencia consiste en el resultado obtenido que, en el caso de un programa con errores será un mensaje informatívo del error cometído. Evitamos, de este modo, tener que introducir mecanismos complejos de detección de errores o tener que adoptar la clásica solución de abortar la ejecución del programa.

4.2.2.- GESTION DE MEMORIA

La Memoria es común a todas las tareas del sistema y en ella se almacenan, tanto el código del programa a ejecutar, como sus datos y resultados. Los problemas básicos, a considerar, en cuanto a la gestión de Memoria son los siguientes:

- La forma en la que las distintas tareas del sistema acceden a la zona de memoria libre, para construir nuevas estructuras.

- La forma en que se realiza la recolecciión, cuando se acaba el espacio de memoria libre.

- La compartición de estructuras por parte de las tareas.

La organización de memoria que proponemos, consiste en la utilización de bloques de tamaño fijo, que se asignan a las tareas conforme éstas lo van necesitando. Cada tarea dispone, de este modo, de una zona propia de memoria, en cuanto a que no debe competir con ninguna otra tarea por las celdas libres que hay en ella.

Ya que nuestro sistema está soportado por un sistema monoprocesador, la Memoria se puede gestionar de forma centralizada. El objetívo perseguido realizando esta asignación de bloques, es que esta gestión sea lo menos gravosa posible, permitiendo que una tarea utilice celdas nuevas con los mínimos requisitos de sincronización con otras tareas.

La labor de recolección se reparte entre las tareas, encargándose cada una de trasladar a su bloque sus celdas accesibles durante la recolección, si bien es verdad, que es necesaria una cierta sincronización, debido a que puede haber varias tareas que compartan las mismas estructuras. Es necesario que una celda sólo se mueva una vez a la zona activa.

Esta asignación de bloques, potencia además la localidad de referencias, aspecto de gran importancia en sistemas con memoria virtual, ya que una tarea realiza sus evaluaciones construyendo las estructuras necesarias dentro de su bloque y, posteriormente, durante la recolección, ya que el "marcado" de las estructuras se realiza por niveles (breadth first), una celda será recolectada por la tarea más cercana a ella, es decir, aquella en la que el camino de acceso a la celda sea más corto.

Por otra parte, la tarea puede acceder, por supuesto, al bloque de memoria de otra tarea, ya que las tareas comparten estructuras comunes como son: expresiones y contextos, que de otro modo deberían estar duplicados en la zona de memoria de cada una de las tareas que las necesite, y en el caso de las recetas, sería necesario actualizarlas una vez evaluadas en cada lugar donde se encontrase. En nuestro caso, es únicamente necesario dotar al sistema de mecanismos que aseguren el acceso exclusívo a esta información.

La tarea que realiza la gestión de la memoria en el sistema, es una tarea especial a la que hemos llamado Gestionador G. Las funciones de

esta tarea consisten en:

- Conceder los bloques de memoria libres a las tareas cuando lo necesiten.

- Ordenar la recolección.

- Detectar que no hay memoria suficiente para poder continuar y abortar la ejecución.

Esta tarea viene definida por la siguiente información:

- Su estado: S_g
- Una cola de mensajes de Gestión: QG_g

El Gestionador puede estar en alguno de los estados: RECOLECCION o CONCESION.

- RECOLECCION:

  Se dice que G está en estado de RECOLECCION, lo que denominaremos como "R", cuando se han acabado los bloques de memoria libres, ha ordenado a las tareas que recolecten su información, y queda alguna que aún no ha acabado de recolectar.

- CONCESION:

  Se dice que G está en estado de CONCESION, lo que denominaremos como "C", cuando su labor consiste en conceder bloques de memoria libres para que las tareas realicen sus evaluaciones, y además no hay ninguna tarea recolectando.

Tanto la concesión de bloques como la ordenación de la recolección, se llevan a cabo por medio de mensajes que se intercambian las tareas que realizan la evaluación del programa y el gestionador, y que se verán en el apartado siguiente, que trata la gestión conjunta de evaluación y memoria.

4.2.3.- GESTION CONJUNTA

Se dedica este apartado, a describir el sistema funcional, desde el punto de vista de la interacción de las tareas que realizan la evaluación del programa, con el gestionador de la memoria del sistema.

Como ya dijimos anteriormente, las tareas necesitan espacio de memoria libre para realizar su labor, y el Gestionador es el encargado de proporcionárselo. Por otra parte, el Gestionador debe detectar el hecho de que se acabe el espacio de memoria libre de que se dispone y ordenar la recolección.

Estas funciones se realizan mediante el paso de un conjunto de mensajes entre las tareas y el Gestionador, que llamaremos mensajes de Gestión, para distinguirlos de los mensajes de Evaluación vistos en el apartado 4.2.1. Estos mensajes constan de información acerca de las tareas o bloques de memoria de que se trata y son los que se indican a continuación. Su existencia está soportada del mismo que aquellos, mediante una cola asociada a cada una de las tareas.

[B. T_i]

Por medio de este mensaje, una tarea T_i, pide un bloque de memoria al Gestionador.

[C. b_i]

Por medio de este mensaje, el Gestionador concede el bloque de memoria b_i a una tarea.

[W]

Mensaje mediante el cual, el Gestionador avisa a una tarea que debe esperar, ya que, no hay más bloques de memoria libres y, por lo tanto, es necesario realizar la recolección.

Es importante destacar, que los mensajes de Gestión son prioritarios a los de Evaluación. La inspección de la cola de mensajes de Gestión, por parte de una tarea, en el momento adecuado es de vital importancia. Si contiene un mensaje, informando de que no hay espacio de memoria libre y

la tarea lo ignora, pasando a ejecutar una instrucción que consuma memoria, esto provocará una nueva petición de bloque, petición que no recibirá contestación y que se habría evitado si, antes de iniciar la ejecución de la instrucción se hubiese inspeccionado esta cola de mensajes. Creemos, en consecuencia, que una tarea debe examinar la cola de mensajes de Gestión de la forma siguiente:

- Antes de la ejecución de cada instrucción, en el caso de tareas ACTIVAS.
- Antes de examinar la cola de mensajes de Evaluación, en el caso de tareas en estado de ESPERA.

Las tareas que estén en cualquier otro estado: LATENTE o MUERTO no precisan inspeccionar esta cola, ya que no hacen ningún uso de la memoria.

En cuanto a las tareas, definimos dos nuevos tipos de estado en los que se puede encontrar una tarea: el estado de la tarea con respecto al sistema, que denominaremos ST_i, y el estado de la tarea con respecto al Gestionador de memoria, que denominaremos SG_i.

Con respecto al sistema, una tarea puede estar en alguno de los estados EVALUACION o RECOLECCION, según esté realizando la labor de evaluar la parte del resultado que le ha sido asignada o recolectando su información (debido a que no existe espacio de memoria libre para continuar).

ST_i = EV significa que T_i está en Evaluación
ST_i = RC significa que T_i está Recolectando

Con respecto al Gestionador de memoria, una tarea puede estar en alguno de los estados: NO PENDIENTE del Gestionador o PARADA PENDIENTE del Gestionador.

- NO PENDIENTE del Gestionador:

  Una tarea está en este estado, que designaremos como NPG, cuando no depende del Gestionador para poder realizar sus evaluaciones, es decir, cuando tiene suficiente espacio libre en su bloque de memoria.

- PARADA PENDIENTE del Gestionador:

Una tarea está en este estado, que designaremos como PPG, cuando depende del Gestionador para poder realizar sus evaluaciones, es decir, está parada en espera de que éste le envíe un bloque de memoria libre, que necesita para poder continuar. En este estado, la tarea no realiza, por lo tanto, ningún tipo de labor.

SG_i = NPG	significa que T_i no necesita un nuevo bloque de memoria para poder continuar
SG_i = PPG	T_i está parada esperando respuesta del Gestionador

Llamando { B } al conjunto de bloques libres de memoria, el sistema funcional S_F en cuanto a gestión conjunta queda definido de la siguiente forma:

$$S_F = < T, G, MG, B >$$

Una tarea T_i queda definida, a efectos de gestión de memoria por la siguiente información:

- Su nombre
- Su estado con respecto al sistema: ST_i
- Su estado con respecto al Gestionador: SG_i
- El último bloque de memoria que le fué asignado: BQ_i
- Su cola de mensajes de Gestión: QG_i

y la representaremos de la forma:

$$T_i(ST_i, SG_i, BQ_i, QG_i)$$

La tarea G queda definida por:

- Su estado: S_g
- Su cola de mensajes de Gestión: QG_g

y la representaremos de la forma:

$$G(S_g, QG_g)$$

Reglas que definen el comportamiento del sistema, en cuanto a disponibilidad de espacio en la Memoria

RG-0

Inicialmente:

$S_g = C$	Gestionador en estado de CONCESION
$ST_i = EV \; \forall T_i \in \{ T \}$	Todas las tareas en Evaluación
$SG_i = NPG \; \forall T_i \in \{ T \}$	No hay ninguna tarea parada por el Gestionador
QG_i = vacía $\forall T_i \in \{ T \}$	Las Colas de Gestión vacías
QG_g = vacía	
Todos los bloques libres	

RG-1

Una tarea T_i, no pendiente del Gestionador, que no tiene espacio libre en su bloque, pide un nuevo bloque a G.

$$T_i(SE_i, NPG, lleno, QG_i) \text{ ----> } T_i(SE_i, PPG, lleno, QG_i) \; \underline{y} \; QG_g \text{ ----> } QG_g:[B.\ T_i]$$

T_i queda Parada, ya que no puede continuar su ejecución.

RG-2

Una tarea T_i, Parada pendiente del Gestionador, recibe un bloque. Entonces continua su ejecución, si estaba en Evaluación. Si, por el contrario, estaba en Recolección realiza su recolección.

$$T_i(SE_i,\ PPG,\ BQ_i,\ [C.b_j]:QG_i) \longrightarrow T_i(SE_i,\ NPG,\ b_j,\ QG_i)$$

RG-3

Una tarea T_i, en Evaluación, recibe un mensaje [W] de G. Esto significa que es preciso recolectar, pero T_i tiene que esperar a que todas las tareas del sistema, excepto G, estén Paradas.

$$T_i(EV,SG_i,\ BQ_i,\ [W]:QG_i) \longrightarrow T_i(RC,\ PPG,\ BQ_i,\ QG_i)$$

RG-4

Una tarea T_i acaba en algún momento de recolectar sus celdas, ya que el número de celdas es finito, pasando de nuevo al estado de Evaluación.

$$T_i(RC,\ NPG,\ BQ_i,\ QG_i) \longrightarrow T_i(EV,\ NPG,\ BQ_i,\ QG_i)$$

RG-5

El Gestionador recibe una petición de bloque, y hay bloques libres en la memoria. Entonces G concede el primero de ellos.

$$G(S_g,\ [B.\ T_i]:\ QG_g)\ \underline{y}\ \exists\, b_j \in \{\ B\ \} \longrightarrow G(S_g,\ QG_g)\ \underline{y}\ QG_i \longrightarrow QG_i:[C.\ b_j]$$

RG-6

El Gestionador, en estado de Concesión, recibe la petición de un bloque y no hay bloques libres en la memoria. Es preciso entonces recolectar. Para ello, es necesario parar previamente todas las tareas.

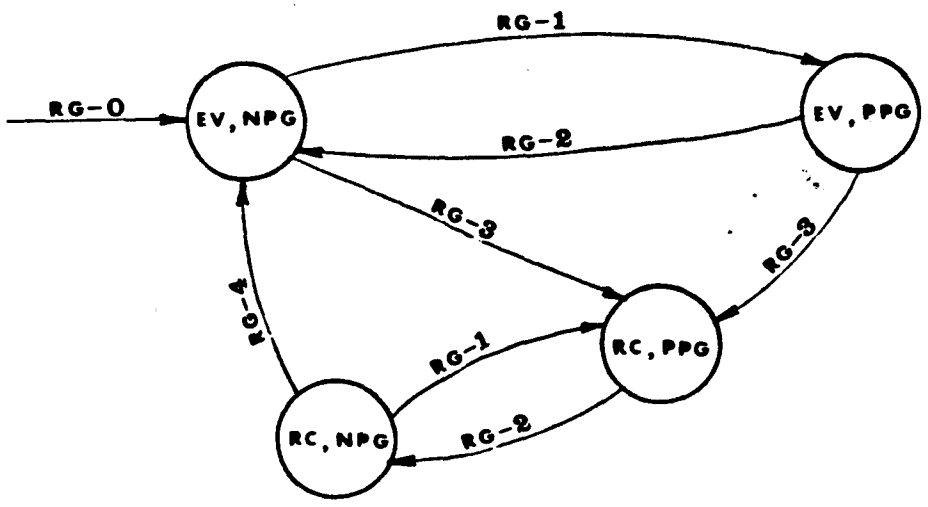
$$G(C,\ [B.\ T_i]:QG_g)\ \underline{y}\ \nexists\, b_j \in \{\ B\ \} \longrightarrow G(R,\ \text{vacía})\ \underline{y}\ \forall T_i \in \{T\}\ QG_i \longrightarrow QG_i:[\ W\ ]$$

RG-7

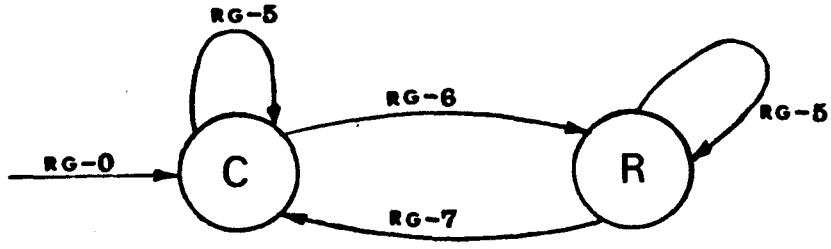
El Gestionador está en estado de Recolección y acaban de recolectar todas las tareas del sistema.

$$G(R, QG_g) \ \underline{y} \ \forall T_i \in \{ T \} \ SG_i = EV \longrightarrow G(C, QG_g)$$

El diagrama de estados de las tareas y el del Gestionador son los que se muestran a continuación:



- Diagrama de estados de las tareas -



- Diagrama de estados del Gestionador -

Como se puede comprobar en las reglas RG-3 y RG-6, el Gestionador antes de realizar una recolección, para al resto de las tareas del sistema. Esto es debido, a que antes de recolectar, cada tarea debe disponer de un bloque libre de memoria para poder comenzar la labor de recolección. De

no hacerlo así, si una de las tareas del sistema tuviera espacio libre en su bloque para poder continuar, el resto de las tareas que dependiesen de la recolección, para seguir sus evaluaciones, deberían esperar a que ésta agotase su bloque. Por otra parte, si la utilización de memoria de esta tarea fuera muy pequeña, el no realizar la recolección podría suponer una excesiva espera para el resto de las tareas.

Una vez están paradas todas las tareas, comienza la recolección. El algorítmo de recolección que hemos utilizado, sigue el modelo de Baker [3] y se caracteriza por disponer de dos espacios de memoria, uno activo y otro pasivo. El Gestionador conmuta entonces de espacio de memoria, y envía a cada una de las tareas un bloque libre del nuevo espacio, para que ellas realicen la recolección.

Con este modelo se reparte la labor de recolección entre las tareas del sistema. Cada tarea es responsable de recolectar las celdas que alguna vez utilizó y ahora son inútiles. Si las tareas no compartiesen estructuras del programa, podrían recolectar de forma independiente. Sin embargo, en nuestro sistema esto no se cumple, ya que hay estructuras que las tareas heredan de quienes las arrancaron y, por tanto, es preciso asegurar el acceso exclusivo a las celdas de memoria que se mueven del espacio pasivo al activo, para evitar que se muevan más de una vez.

El conjunto de reglas, expuestas anteriormente, contempla todos los casos que pueden darse en el sistema y, por lo tanto, aseguran su completitud. No se contemplan pues los casos imposibles que describimos a continuación:

Caso 1.-

Una tarea T_i, en estado PPG, no puede pedir un bloque libre de memoria.

Caso 2.-

Una tarea T_i, en estado NPPG, no puede recibir un bloque.

Caso 3.-

Una tarea T_i en estado de Recolección, no puede recibir un mensaje de recolección.

Demostraremos a continuación que no se puede dar ninguno de estos casos.

Caso 1.-

demostración

Una tarea T_i se Para cuando pide un bloque, aplicando RG-1.

El estado de Parada asegura que T_i no realiza ningún tipo de evaluación, hasta que pase al estado de NO PENDIENTE del Gestionador NPG. Por lo tanto, si T_i no realiza ninguna evaluación, no puede verse en la necesidad de pedir un bloque de memoria libre.

Caso 2.-

demostración

Inicialmente, todas las tareas están en estado NPPG y todas las colas de Gestión están vacías, por la regla RG-0.

Puede ocurrir:

a) Que T_i nunca haya estado Parada, en cuyo caso no habrá pedido bloque alguno y, por lo tanto, ya que solo puede recibirlo aplicando la regla RG-2, no hay posibilidad de que lo reciba.

b) Que T_i haya estado Parada. Unicamente ha podido pasar al estado PPG aplicando las reglas RG-1 y RG-3. La regla RG-2 asegura que T_i recibe el bloque que esperaba.

Caso 3.-

demostración

Un mensaje de recolección [W] sólo aparece cuando el Gestionador está en estado de Concesión y se han acabado los bloques de memoria libres, aplicando la regla RG-6.

El Gestionador sólo puede estar en estado de Concesión si todas las tareas han acabado de recolectar y, por tanto, estarán en estado de Evaluación.

Ya que en nuestro sistema, cuando una tarea acaba la recolección de su información continúa su evaluación independientemente del resto de las tareas, que una tarea T_i en estado de Recolección recibiese un mensaje [W], significaría que para acabar de recolectar es preciso recolectar. Ello podría ser debido, por ejemplo, a que una de las tareas que ya acabó de recolectar T_j, consume, en su evaluación, memoria más rápidamente de lo que lo hace T_i para recolectar. No está muy claro, sin embargo, que si se hubiese retrasado T_j, T_i pudiera haber acabado su recolección y, por lo tanto, continuarían ambas en Evaluación.

Ante este hecho, en la realización de nuestro sistema, hacemos que el Gestionador pueda detectar este caso y decidimos abortar el sistema en lugar de ordenar una nueva recolección.

Está asegurada también la finitud de la recolección. El número de celdas que debe recolectar cada tarea es finito, ya que lo es el número de celdas de la Memoria. La regla RG-4 asegura que las tareas continuarán su evaluación una vez terminada la recolección.

CAPITULO 5

EVALUACIONES

5. EVALUACIONES

Este capítulo está dedicado a la evaluación de los modelos de representación de información, que planteamos y desarrollamos en profundidad en el capítulo 3. Hemos de diferenciar dos aspectos a evaluar, la utilización de la representación lineal para el código a ejecutar por la máquina, y la utilización de la estructura "vector" para representar el contexto de evaluación de un programa.

Los datos que nos sirven de base para estas evaluaciones se han obtenido, ejecutando cinco programas funcionales prototipo en tres modelos de máquina virtual diferentes.

Los tres modelos de máquina virtual han sido simulados mediante programas escritos en lenguaje C, en el VAX 11/750 del Centro de Cálculo de la Facultad de Informática de Madrid y son los siguientes:

- S1O: Sistema funcional cuya única estructura de representación interna es la de listas, y cuyo inmediato antecesor es S7 (Mañas [83]).

- SLO: Sistema funcional en el que coexisten la representación lineal para el código y la de listas para el resto de la información.

- SVO: Sistema funcional en el que se introduce la estructura "vector" para la representación del contexto.

Los programas funcionales utilizados como prototipo se muestran en el apéndice 2 y algunos de ellos han sido ya utilizados por otros autores como prototipos para la evaluación de diversos aspectos de los sistemas funcionales (Turner [79], Mañas [83], Henderson [28]). Estos programas cubren en gran medida el rango de aplicaciones del sistema y son los siguientes:

- NORM representa a los programas con gran cantidad de expresiones aritméticas. Calcula números aleatorios distribuidos normalmente.

- PERM representa a los programas dedicados al manejo de listas. Calcula las permutaciones de los átomos de una lista.

- PRIMOS y ROUND representan a los programas que manejan listas potencialmente infinitas. PRIMOS calcula la lista de los números primos utilizando la criba de Eratostenes y ROUND calcula la lista ordenada de los números cuyos únicos factores primos son 2, 3 o 5.

- NFIB calcula el número de aplicaciones necesarias para obtener el n-ésimo elemento de la serie de Fibonacci. En este programa proliferan tanto las aplicaciones de función como las operaciones aritméticas.

Este capítulo está dividido en tres apartados. En el primero, analizamos y evaluamos la representación lineal para el código de máquina. El segundo se dedica a evaluar las repercusiones de las optimizaciones de código en el sistema. Por último, en el tercer apartado estudiamos y evaluamos las ventajas que proporciona la utilización de vectores para representar el contexto de evaluación.

5.1.-REPRESENTACION LINEAL DEL CODIGO

Tratamos en este apartado, las repercusiones de la representación lineal del código, sobre aspectos tales como velocidad de ejecución, ocupación de memoria, y no recolección de la zona de código.

Este tipo de representación para el código de máquina proporciona, en principio, una serie de ventajas sobre la representación de listas entre las que cabe destacar:

- La mejora del tiempo de ejecución de los programas.
- La utilización de un espacio menor de Memoria.
- La disminución del tiempo utilizado para la recolección.

Pese a estas ventajas, presenta el inconveniente de que, al permanecer el código fijo en memoria no hay posibilidad de recolectarlo.

Es preciso evaluar, hasta qué punto las ventajas señaladas introducen una mejora significativa en el sistema, y si compensan el inconveniente de esta representación. En lo que sigue, evaluaremos cada uno de estos puntos.

5.1.1.- VELOCIDAD DE EJECUCION

La utilización de la representación lineal para el código, reduce el tiempo de ejecución de un programa ya que, el acceso al código se realiza utilizando un registro contador de programa, que se incrementa para acceder a la siguiente instrucción (salvo en los casos de salto), en lugar de hacerlo siguiendo apuntadores. Se ha observado el tiempo (en segundos) de CPU, aproximado, utilizado en la ejecución de cada uno de los programas prototipo sobre S10 (código de listas) y SL0 (código lineal) cuyos resultados se muestran en la tabla 1.

Programa	S10	SL0	Ahorro
PRIMOS	37.8	35.4	6.4%
ROUND	90.3	55.5	38.5%
NFIB	90.5	86.6	4.3%
NORM	126.3	103.3	18.2%
PERM	113.9	98.5	13.5

- tabla 1 - Tiempo de Ejecución

5.1.2.- OCUPACION DE MEMORIA

Se ha considerado necesario, un estudio del ahorro de memoria que supone el almacenamiento del código en posiciones consecutívas, ya que en la representación de listas, existen técnicas de linealización como el "cdr-coding", a las que nos referimos en el capítulo de antecedentes, que persiguen este mismo fin, y tal vez podrían proporcionarnos un ahorro similar.

La tabla 2 muestra la ocupación inicial del código, medida en número de bits, en el modelo de representación de listas y en el modelo de representación lineal. La representación de listas utiliza como tamaño básico

de memoria 48 bits, que se emplean para representar los tipos del "car" y "cdr" (8 + 8 bits) y las direcciones o valores de éstos (16 + 16 bits), mientras que la representación lineal utiliza palabras consecutívas de 16 bits para representar un código de operación, un dato inmediato o una dirección.

	listas	lineal	% ahorro
NORM	10848	4112	62%
PERM	11952	4752	60%
PRIMOS	8160	3104	62%
NFIB	2304	848	63%
ROUND	11280	4304	62%

- tabla 1 - Ocupación inicial del código

A la vista de estos resultados se comprueba, que el ahorro que supone la utilización de esta representación, en cuanto a ocupación de memoria, con respecto a una representación de listas no linealizada, es bastante significatívo, obteniéndose desde un 60% para PERM hasta un 63% para NFIB.

Queda únicamente por comprobar, si se conseguiría este ahorro utilizando una técnica de linealización de listas , como por ejemplo "cdr-coding". En el mejor de los casos, cuando pudiera linealizarse toda la lista, son necesarios 32 bits para representar, los tipos del "car" y el "cdr" y la dirección del "car". Aún así, para representar N instrucciones necesitaríamos 32*N bits, mientras que en la representación lineal, solamente serían necesarios 16*N bits, lo que supone un ahorro del 50% de la zona de código a favor de la representación lineal.

5.1.3.-REPERCUSIONES SOBRE LA RECOLECCION

Una consecuencia inmediata de la representación lineal del código máquina, es el hecho de que éste permanece fijo en memoria a lo largo de la ejecución del programa (del mismo modo que sucede en las arquitecturas convencionales) y, por lo tanto, se pierde la oportunidad de hacer reutiliza-

ble esta zona de la memoria, o lo que es lo mismo, se pierde la oportunidad de recolectar el código.

Es preciso evaluar, basándonos en este hecho, si es adecuada o no esta representación. Debemos contestar a preguntas como Qué cantidad de código se abandona realmente a lo largo de la ejecución de un programa (se "destruye"), cuando se utiliza la representación de listas?. Aumenta el número de recolecciones necesarias para ejecutar un programa, manteniendo el código fijo en memoria?. Una respuesta afirmativa a esta última pregunta supondría un aumento en el tiempo total de ejecución. En cualquier caso, cabe preguntarse si queda compensado por el ahorro de espacio conseguido con la utilización de la representación lineal.

Estudiaremos, en primer lugar, las características de los programas utilizados como prototipo, en cuanto al uso que hacen de la memoria a lo largo de su ejecución. Las tablas 3 a 7 recogen el comportamiento de cada uno de ellos, en lo referente al número de celdas de memoria accesibles. Se han tomado las medidas, cada 60000 instrucciones ejecutadas, bajo dos condicionantes: primero, si se permite recolectar código, y segundo, si no se permite, permaneciendo fijo en memoria a lo largo de la ejecución del programa. Hemos utilizado en estas tablas la siguiente notación:

NCA_{cr} número de celdas accesibles con recolección de código
NCA_{sr} número de celdas accesibles sin recolección de código
%INST porcentaje de instrucciónes ejecutadas hasta el momento con respecto al número total de instrucciones ejecutadas

Los datos de estas tablas quedan reflejados gráficamente en la figura 1, que representa la memoria accesible, a lo largo de la ejecución de cada uno de los programas prototipo. En esta figura sólo están representadas las gráficas correspondientes a las celdas de memoria accesibles sin recolección de código, debido a que la escasa diferencia existente entre NCA_{sr} y NCA_{cr} no es representable.

PERM	NCA_{cr}	NCA_{sr}	%INST
	590	626	0.18
	646	682	0.36
	767	803	0.54
	891	927	0.72
	396	432	0.9

- tabla 3 -

PRIMOS	NCA_{cr}	NCA_{sr}	%INST
	908	951	0.19
	995	1038	0.38
	1384	1427	0.56
	1332	1375	0.75
	1916	1959	0.94

- tabla 4 -

NFIB	NCA_{cr}	NCA_{sr}	%INST
	85	97	0.16
	97	107	0.32
	87	99	0.48
	89	101	0.64
	93	105	0.81
	93	105	0.97

- tabla 5 -

NORM	NCA_{cr}	NCA_{sr}	%INST
	1557	1642	0.047
	2855	2940	0.094
	4162	4247	0.14
	5513	5598	0.19
	6821	6906	0.23
	8127	8212	0.28
	9469	9554	0.33
	10784	10869	0.38
	12089	12174	0.42
	13425	13510	0.47
	14749	14834	0.52
	16054	16139	0.57
	17381	17466	0.61
	18715	18800	0.66
	20017	20102	0.71
	21337	21422	0.75
	22682	22767	0.80
	23980	24065	0.85
	25295	25380	0.90
	26649	26734	0.94
	27943	28028	0.99

- tabla 6 -

ROUND	NCA_{cr}	NCA_{sr}	%INST
	1484	1586	0.16
	3334	3436	0.32
	5209	5311	0.48
	7075	7177	0.64
	8948	9050	0.80
	10631	10733	0.96

- tabla 7 -

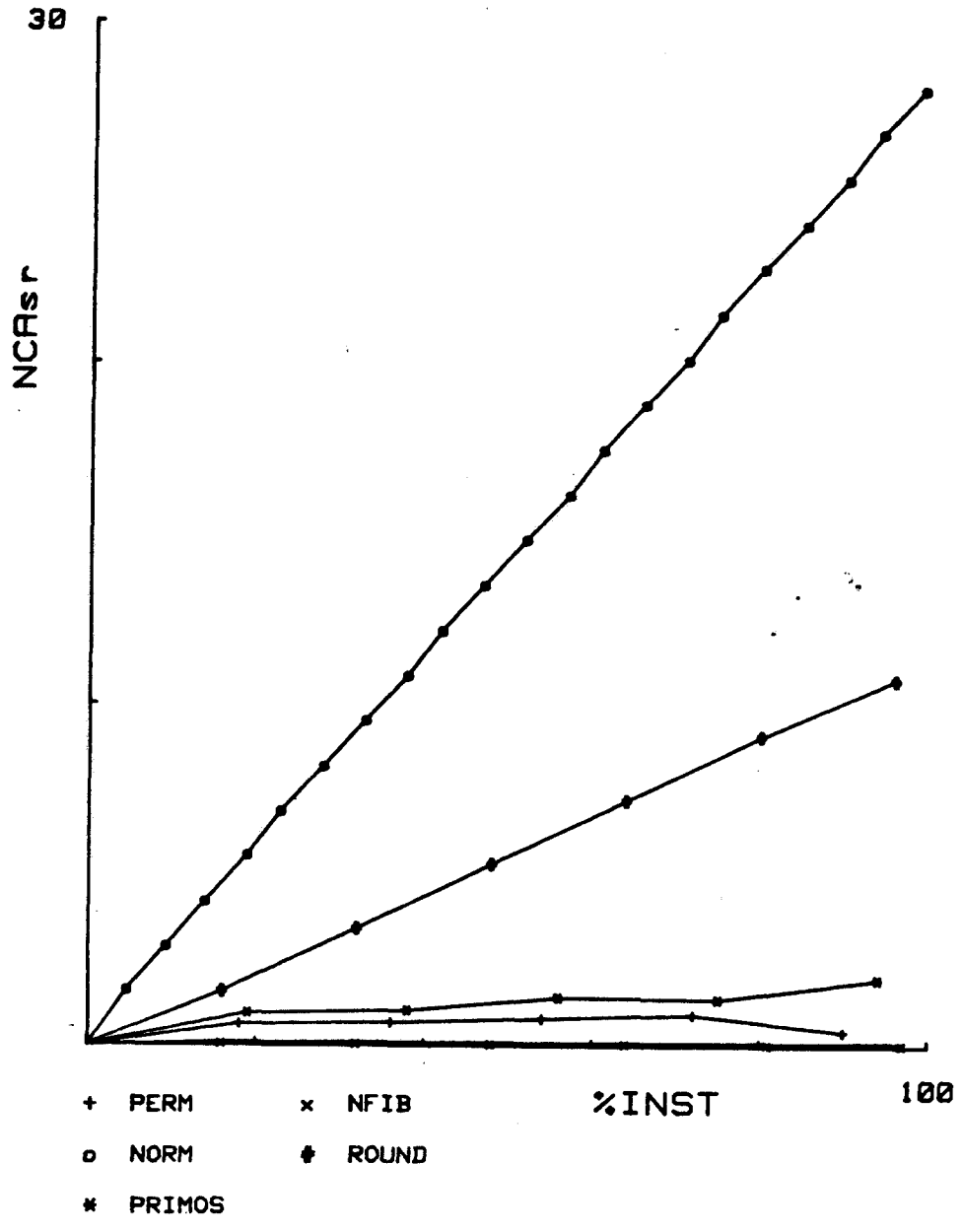


Figura 1.- Memoria accesible durante la ejecución

Como se puede comprobar en esta figura, el mayor número de celdas accesibles corresponde al programa NORM que llega hasta 28028 celdas poco antes de finalizar su ejecución, y el menor a NFIB, con sólo un máximo de 105 celdas accesibles. Utilizaremos estos datos en lo que sigue, para comprobar qué fracción de código se recolecta utilizando la representación de listas, y qué ahorro se consigue recolectando esta información.

Destrucción de código

En principio, cabe pensar que a lo largo de la ejecución de un programa existen instrucciones que una vez ejecutadas ya no vuelven a utilizarse. En concreto, en cuanto a las recetas, se podría pensar que una vez evaluadas, su código deja de ser útil y por lo tanto se puede recolectar. Es preciso evaluar qué cantidad de código deja de ser útil, realmente, a lo largo de la ejecución, antes de afirmar que el dejar el código fijo en memoria no repercute negativamente en la utilización de la memoria.

Las gráficas de la figura 2, representan el porcentaje de celdas de código accesibles, con respecto a la ocupación de código inicial, medido cada 60000 instrucciones ejecutadas. En estas graficas se recogen las tablas 8 a 12, en las que se ha utilizado la notación siguiente:

- NCCA número de celdas de código accesibles.
- NCCI número de celdas que ocupa el código inicialmente

PERM	NCCA	NCCA/NCCI	%INST
	249	1	0
	218	0.87	0.18
	218	0.87	0.36
	218	0.87	0.54
	218	0.87	0.72
	218	0.87	0.90

- tabla 8 -

PRIMOS	NCCA	NCCA/NCCI	%INST
	170	1	0
	132	0.78	0.16
	132	0.78	0.32
	132	0.78	0.48
	132	0.78	0.64
	132	0.78	0.81
	132	0.78	0.97

- tabla 9 -

NFIB	NCCA	NCCA/NCCI	%INST
	170	1	0
	41	0.85	0.19
	41	0.85	0.38
	41	0.85	0.56
	41	0.85	0.75
	41	0.85	0.94

- tabla 10 -

ROUND	NCCA	NCCA/NCCI	%INST
	235	1	0
	138	0.59	0.16
	138	0.59	0.32
	138	0.59	0.48
	138	0.59	0.64
	138	0.59	0.80
	138	0.59	0.96

- tabla 11 -

NORM	NCCA	NCCA/NCCI	%INST
	226	1	0
	146	0.65	0.047
	146	0.65	0.094
	146	0.65	0.14
	146	0.65	0.19
	146	0.65	0.23
	146	0.65	0.28
	146	0.65	0.33
	146	0.65	0.38
	146	0.65	0.42
	146	0.65	0.47
	146	0.65	0.52
	146	0.65	0.57
	146	0.65	0.61
	146	0.65	0.66
	146	0.65	0.71
	146	0.65	0.75
	146	0.65	0.80
	146	0.65	0.85
	146	0.65	0.90
	146	0.65	0.94
	146	0.65	0.99

- tabla 12 -

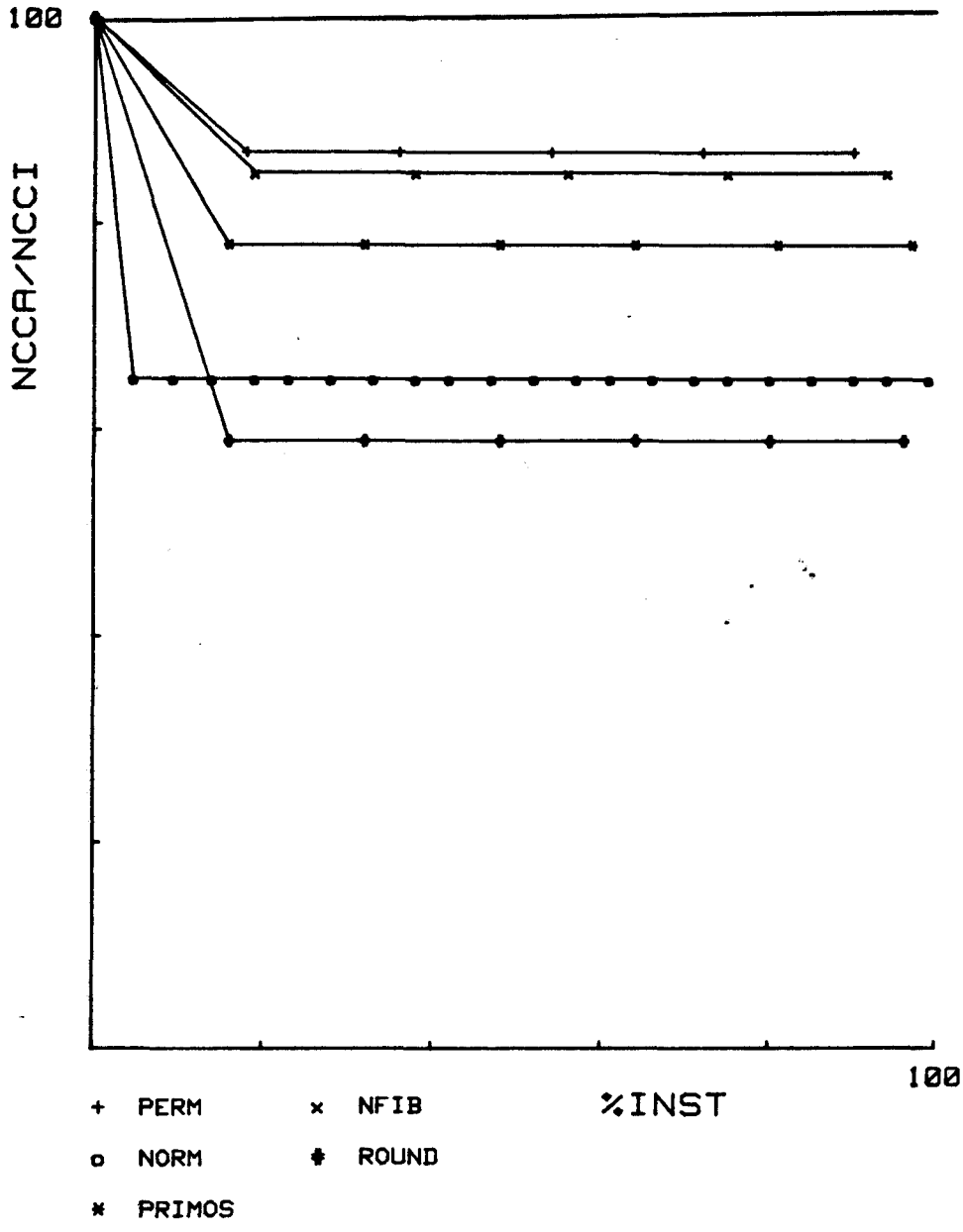


Figura 2.- Porcentaje de código inicial accesible

La línea recta, correspondiente a NCCA = NCCI, representa el porcentaje de código accesible cuando se mantiene el código fijo en memoria: todo el código inicial se mantiene, y por lo tanto no existe destrucción alguna. La máxima distancia entre cada una de las restantes gráficas y esta línea representa el porcentaje de código inicial que se destruye.

A la vista de estos resultados se puede asegurar, que el abandono de código se produce prácticamente en su totalidad en los primeros instan-

tes de la ejecución, afectando principalmente al número de celdas liberadas en la primera recolección (caso de existir ésta), pasando posteriormente a una fase de estabilización, en la que la zona de código accesible permanece constante hasta finalizar la ejecución.

Los porcentajes de cantidad de código abandonado, que van desde un 13% hasta un 41% sobre la ocupación inicial, nos indican la fracción de esta zona de memoria desperdiciada durante la mayor parte del tiempo. Si comparamos estos porcentajes con el ahorro de memoria obtenido, en cuanto a ocupación del código, que como vimos en 5.1.1 van desde un 60% hasta un 63%, el resultado está claramente a favor de la utilización de la representación lineal.

Es más significativo, sin embargo, evaluar la fracción de memoria que no es posible reutilizar, con respecto al total de celdas accesibles, que incluyen código y datos, ya que es este espacio el que realmente se utiliza. Esta fracción está representada en las gráficas de la figura 3. No se han incluido las tablas correspondientes, debido a que su contenido es directamente deducible de los datos de tablas anteriores:

$$(NCCI-NCCA)/NCA_{sr}$$

En esta figura, se observa, que el porcentaje de memoria que se desperdicia, manteniendo el código fijo, es como máximo un 7% de toda la zona accesible, en el caso del programa PERM, y que es menor cuanto mayor sea el número total de celdas accesibles.

Por último, hemos representado en las gráficas de la figura 4 el ahorro de memoria que supone la recolección de código, realizando la recolección cada 60000 instrucciones ejecutadas. Esta figura recoge los datos de las tablas 13 a 17, donde se ha utilizado la siguiente notación:

CL_{cr} número de celdas libres con recolección de código
CL_{sr} número de celdas libres sin recolección de código

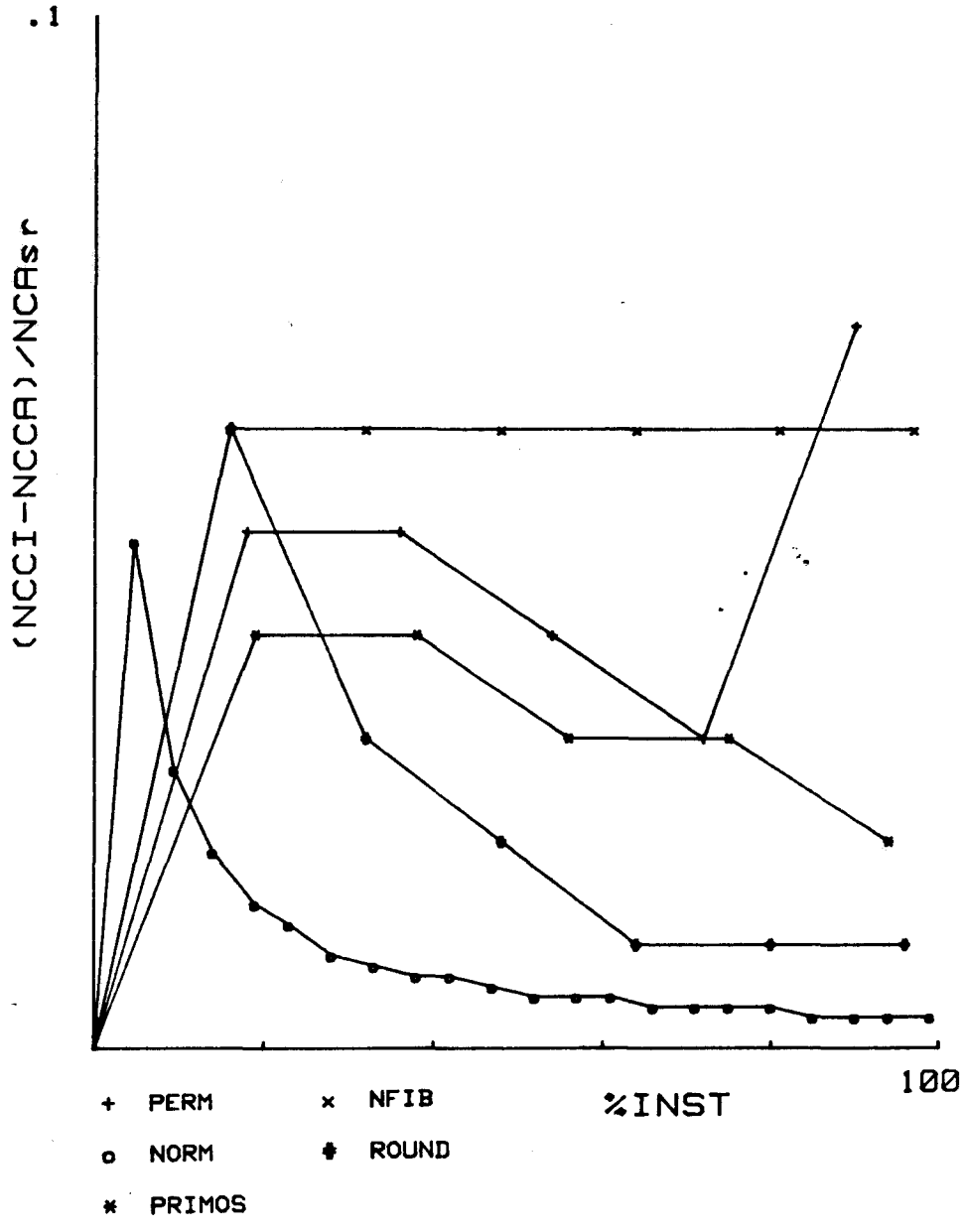


Figura 3.- Código abandonado con respecto al total de celdas accesibles.

PERM	CL_{cr}	CL_{sr}	$(CL_{cr} - CL_{sr})/CL_{cr}$	%INST
	64946	64910	0.0006	0.18
	64890	64854	0.0006	0.36
	64769	64733	0.0006	0.54
	64645	64609	0.0006	0.72
	65140	65104	0.0006	0.90

- tabla 13 -

PRIMOS	CL_{cr}	CL_{sr}	$(CL_{cr} - CL_{sr})/CL_{cr}$	%INST
	64628	64585	0.001	0.19
	64541	64498	0.001	0.38
	64152	64109	0.001	0.56
	64202	64161	0.001	0.75
	63620	63577	0.001	0.94

- tabla 14 -

NFIB	CL_{cr}	CL_{sr}	$(CL_{cr} - CL_{sr})/CL_{cr}$	%INST
	65451	65439	0.002	0.16
	65439	65429	0.002	0.32
	65449	65437	0.002	0.48
	65447	65435	0.002	0.64
	65443	65431	0.002	0.81
	65443	65431	0.002	0.97

- tabla 15 -

NORM	CL_{cr}	CL_{sr}	$(CL_{cr} - CL_{sr})/CL_{cr}$	%INST
	63979	63894	0.001	0.047
	62681	62596	0.001	0.094
	61374	61289	0.001	0.14
	60023	59938	0.001	0.19
	58715	58630	0.001	0.23
	57409	57324	0.001	0.28
	56067	55982	0.002	0.33
	54752	54667	0.002	0.38
	53447	53362	0.002	0.42
	52111	52026	0.002	0.47
	50787	50702	0.002	0.52
	49482	49397	0.002	0.57
	48155	48070	0.002	0.61
	46821	46736	0.002	0.66
	45519	45434	0.002	0.71
	44199	44114	0.002	0.75
	42854	42769	0.002	0.80
	41556	41471	0.002	0.85
	40241	40156	0.002	0.90
	38887	38802	0.002	0.94
	37593	37508	0.002	0.99

- tabla 16 -

ROUND	CL_{cr}	CL_{sr}	$(CL_{cr} - CL_{sr})/CL_{cr}$	%INST
	64052	63950	0.001	0.16
	62202	62100	0.002	0.32
	60327	60225	0.002	0.48
	58461	58359	0.002	0.64
	56588	56486	0.002	0.80
	54905	54803	0.002	0.96

- tabla 17 -

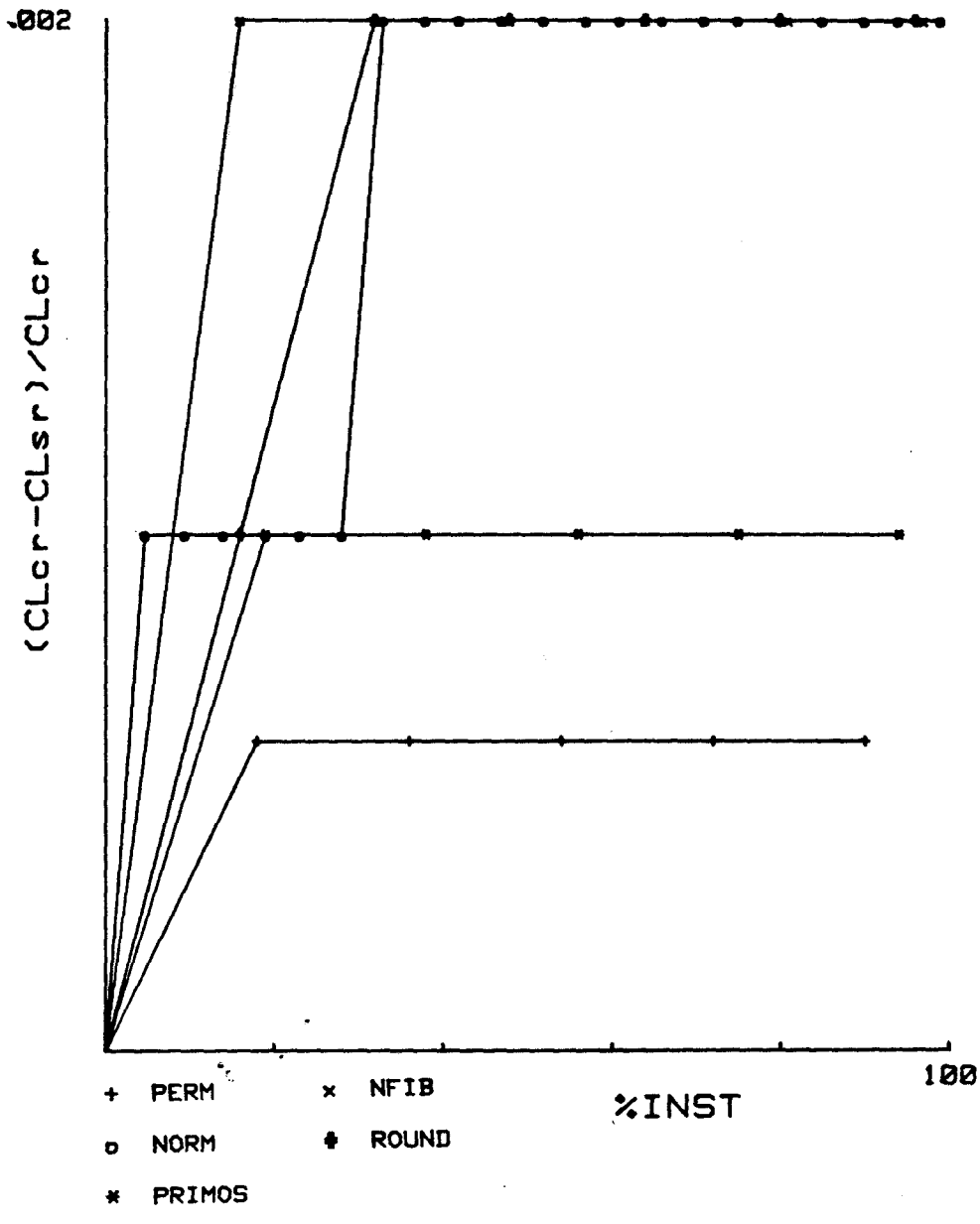


Figura 4.- Ahorro de memoria con recolección de código

Como se puede observar en esta figura, la cantidad de memoria que se consigue reutilizar, recolectando código, representa en el mejor de los casos únicamente un 0.2% del total de la zona de memoria libre.

Se deduce de todo lo anterior, que el dejar el código fijo a lo largo de la ejecución del programa, no tiene repercusiones negatívas en la utilización de la memoria, ya que si bien es verdad que se pierde la oportunidad de recolectar esa zona de memoria, el ahorro que se consigue con

esta recolección es insignificante, puesto que hay poco código que deje de ser útil. Por otra parte, el ahorro de memoria conseguido, utilizando una representación lineal para el código, compensa la mínima pérdida que supone el no poder recolectarlo.

Número de Recolecciones

La tabla 18 muestra el número de recolecciones necesarias, durante la ejecución de los programas prototipo en los sistemas S10 y SL0, es decir, con y sin recolección de código, NR_{cr} y NR_{sr} respectivamente. Para confeccionar esta tabla, se ha variado la lista de entrada de PERM, de 6 a 7 átomos, con el fin de conseguir un mayor número de recolecciones y poder comprobar si existía alguna variación entre NR_{cr} y NR_{sr}.

programa	NR_{cr}	NR_{sr}
NORM	10	10
PERM	25	25
PRIMOS	2	2
NFIB	1	1
ROUND	3	3

- tabla 18 - Número de recolecciones

A la vista de estos resultados podemos afirmar, que en general no existe variación en cuanto al número de recolecciones necesarias durante la ejecución de un programa. Sólo existirá variación cuando el número de recolecciones sea muy grande, pero esta variación será tan pequeña que no afectará prácticamente al tiempo total de ejecución. En el caso de los programas utilizados como prototipo, no existe variación alguna.

Tiempo de Recolección

Para hacer una estimación del tiempo de recolección y su relación con el número de celdas accesibles en el momento de la recolección, se ha realizado un programa que se caracteriza porque el número de celdas accesibles, a lo largo de toda su ejecución, es proporcional a su argumento de

entrada.

```
sea-rec f (x)

        x = for (1, N)

    y f = función (y)
                si y = NIL ent f (y)
                si no f (-1 (y))

    y for = función (a, b)
                si a > b ent NIL
                si no a : for (a+1, b)
```

Número de celdas accesibles NCA = N+4(N-1)+90

Variando N y midiendo el tiempo de CPU utilizado en la recolección, para cada uno de los valores, se ha comprobado que el tiempo de recolección T_r es linealmente dependiente del número de celdas accesibles en el momento de la recolección.

$$T_r = K.NCA$$

donde K es el tiempo empleado en marcar y mover una celda accesible a la zona activa.

Ya que la representación lineal del código trae consigo la no recolección de la zona de memoria que ocupa, esto significa que el recolector deberá identificar un número menor de celdas accesibles que en el caso de tener que recolectarlo (representación de listas), con lo cual el tiempo del proceso de recolección disminuirá tanto mas cuanto mayor sea la zona de memoria ocupada por el código del programa.

5.2.-OPTIMIZACIONES DE CODIGO

Como comentábamos anteriormente, en un sistema funcional, el papel de cualquier optimización es de gran importancia, ya que revierte en una

mejora de velocidad de ejecución, siempre bienvenida en tales sistemas que pecan en general de ser relatívamente lentos.

Se dedica este apartado a la evaluación de las optimizaciones de código expuestas en el capítulo 3. Estas optimizaciones merecen en principio un estudio diferente, ya que su finalidad tambien lo es. La primera de ellas consiste en no guardar direcciones de retorno inútiles, debe afectar directamente al tamaño del programa y, por lo tanto, a la ocupación de memoria. La segunda consiste en no guardar contextos que no vayan a ser utilizados posteriormente, y deberá influir en el número de celdas accesibles en cada momento. Tienen, sin embargo, una característica común que consiste en que la carga de la pila de estado D debe ser menor, ya que deberá guardarse en ella menos información.

5.2.1.- OPTIMIZACION xRET

Esta optimización consiste básicamente en crear una versión optimizada de las instrucciones LD, CAR, CDR, AP e IMP, cuando van seguidas de la instrucción RET (LDRET, CARRET, CDRRET, APRET, IMPRET), evitándose de esta forma tener que guardar y recuperar una dirección de retorno totalmente inútil en y desde la pila D. Proporciona una serie de ventajas entre las que caben destacar:

- La reducción del código del programa.
- La disminución del tiempo total de ejecución.

El ahorro que supone la utilización de estas optimizaciones, en cuanto a ocupación de memoria se refiere, queda reflejado en la tabla 19 en la que se muestra la ocupación del código, con y sin optimización, en número de palabras de 16 bits. Este ahorro es debido a la sustitución de dos instrucciónes (... y RET) por una sola (...RET) y varía desde un 1.8% para NFIB (programa con pocas instrucciones y por consiguiente pocas optimizaciones), hasta un 11% para PERM y ROUND.

programa	ocupación s.o	ocupación c.o	%ahorro
PERM	306	272	11%
NORM	257	239	7%
PRIMOS	203	184	9%
NFIB	54	53	1.8%
ROUND	274	244	11%

- tabla 19 - Ocupación de código sin y con optimización

Por otro lado, en el apartado 3.2.2, comprobamos que las instrucciones mencionadas más arriba, no siempre admiten optimización, debido a que a veces no tienden a guardar información en la pila D.

Las instrucciones que la admiten son:

- LDRET, CARRET y CDRRET al encontrar una receta sin evaluar.
- APRET al encontrar una función o un selector distinto de cero.
- IMPRET al encontrar una lista.

y las que no la admiten, y por tanto se complican, ya que deben realizar ellas mismas la función de RET son:

- LDRET y CARRET al encontrar un valor.
- CDRRET al encontrar un valor o una promesa de fichero.
- APRET al encontrar un selector igual a cero.
- IMPRET al encontrar un átomo.

En base a esto, veamos, qué proporción de las instrucciones optimizadas admiten tal optimización (NIao) y qué proporción no la admiten (NIna), a lo largo de la ejecución de cada uno de los programas. La tabla 20 muestra los resultados obtenidos.

	PERM		NORM		PRIMOS		ROUND		NFIB	
	NIao	NIna	NIao	NIna	NIao	NIna	NIao	NIna	NIao	NIna
LDRET	262	119	2	4	861	150	3	11323	0	0
APRET	372	0	6999	0	13678	0	13325	0	1	0
CARRET	58	194	3000	499	300	11952	1999	3970	0	0
CDRRET	94	117	998	2998	12664	0	8643	4675	0	0
IMPRET	96	25	1000	1	300	1	4000	1	0	1
Total	882	455	11999	3502	27803	12103	27970	19969	1	1
%	66%	34%	77%	23%	70%	30%	58%	42%	50%	50%

- tabla 20 - Instrucciones que admiten y no admiten optimización

Según estos resultados, se observa que, el porcentaje total de instrucciones que admiten la optimización es superior al de instrucciones que no la admiten, variando desde un 50% para NFIB hasta un 77% para NORM. Ello nos permite afirmar que el tiempo de ejecución disminuirá ya que hay menos instrucciones que ejecutar y las versiones optimizadas deben hacer dos accesos menos a la pila D que las no optimizadas (las correspondientes a guardar el contexto y la dirección de retorno).

En cuanto a instrucciones concretas, existen, desde los casos en los que el porcentaje de las que admiten la optimización es mayor del que no las admiten, siendo el ejemplo más extremo el de APRET con practicamente un 100% (hay que tener en cuenta la escasa probabilidad de aparición de un selector nulo en un programa), hasta los casos inversos, cuyo ejemplo mas claro es el de CARRET. Existen entre ambos extremos instrucciones que se inclinan a un lado o a otro, dependiendo de la aplicación concreta, como por ejemplo CDRRET, que en los programas PRIMOS y ROUND, pertenece al primer grupo, y en PERM y NORM al segundo.

De cualquier forma, el hecho de que alguna de estas instrucciones no admita la optimización, no repercute negativamente en el sistema. La complicación introducida en estas instrucciones, al tener que realizar ellas mismas la función de RET puede realizarse, teniendo una máquina mi-

croprogramada, con un simple salto a la zona de microcódigo de la instrucción RET. Esto supone una ventaja, ya que se elimina la necesidad de tener dos instrucciones, ... y RET, evitándose incrementar el contador de programa e ir a buscar a memoria la siguiente instrucción (RET).

5.2.2.- OPTIMIZACION xNE

Este tipo de optimización consiste en no guardar contextos inútiles en la pila de estado. Sucede cuando éstos contextos no van a ser utilizados posteriormente y por lo tanto no hay por qué guardarlos.

El abandono de este tipo de información deberá repercutir en la cantidad de celdas accesibles en cada instante y como consecuencia en la recolección de restos. Se espera, en principio, un incremento de la zona de memoria libre después de cada recolección. Las tablas 21 a 25 muestran el número de celdas libres, después de cada recolección, efectuada cada 60000 instrucciones ejecutadas, con y sin optimización, NCL_{co} y NCL_{so} respectívamente. Estos resultados quedan reflejados gráficamente en la figura 5, que representa el ahorro de memoria conseguido, al abandonar contextos no útiles, a lo largo del programa.

PERM	NCL_{so}	NCL_{co}	%INST
	54494	54496	0.18
	54445	54445	0.37
	54235	54235	0.56
	54132	54134	0.74
	54566	54569	0.93

- tabla 21 -

NORM	NCL_{so}	NCL_{co}	%INST
	53518	53510	0.09
	52151	52149	0.19
	50779	50777	0.29
	49416	49414	0.39
	48082	48080	0.48
	46723	46721	0.58
	45363	45361	0.68
	43994	43992	0.77
	42619	42617	0.87
	41267	41265	0.96

- tabla 22 -

PRIMOS	NCL_{so}	NCL_{co}	%INST
	54203	54253	0.20
	54068	54068	0.39
	53808	53808	0.59
	53459	53459	0.78
	53558	53558	0.98

- tabla 23 -

NFIB	NCL_{so}	NCL_{co}	%INST
	55088	55100	0.17
	55095	55109	0.34
	55087	55105	0.51
	55090	55104	0.68
	55088	55102	0.85

- tabla 24 -

ROUND	NCL_{so}	NCL_{co}	%INST
	53764	53764	0.24
	52661	52661	0.48
	51678	51678	0.71
	50690	50690	0.95

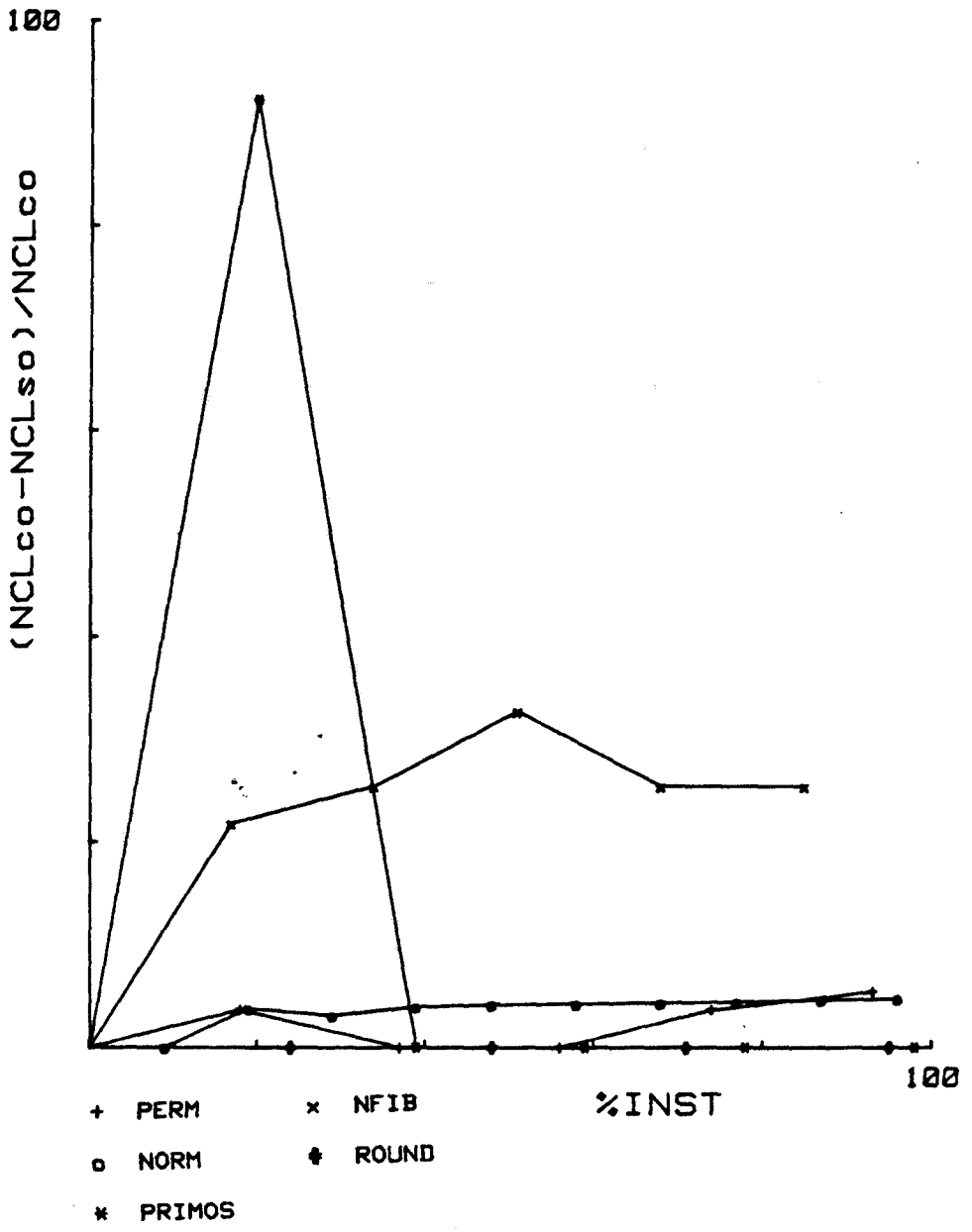


Figura 5.- Ahorro de memoria con la optimización xxNE

Se observa que el ahorro de memoria que supone esta optimización es muy pequeño, como máximo aproximadamente un 1% del total. Ello es debido a

que hay pocos contextos que realmente dejen de ser útiles, puesto que los programas constan, en su mayor parte, de definiciones recursívas y esto hace que prácticamente todos los valores del contexto sean necesarios durante la ejecución de los programas.

5.2.3.- CARGA DE LA PILA DE ESTADO D

Ambas optimizaciones afectan directamente a la carga de la pila D y por lo tanto, al número de accesos a esta pila a lo largo de la ejecución del programa. Recordemos que:

- LD, CAR y CDR acceden a ella para guardar contexto, dirección de retorno y dirección de receta en evaluación, al encontrar una receta.
- AP accede para guardar contexto y dirección de retorno cuando encuentra una función, y sólo dirección de retorno cuando encuentra un selector distinto de cero.
- IMP accede para guardar dirección de retorno cuando encuentra una lista normal o de tipo error.

Evaluaremos a continuación la ocupación máxima de la pila D, con diferentes variaciones en cuando al código del programa se refiere:

- Con la optimización de abandono del contexto exclusivamente (instrucciones ...NE).
- Con la optimización de eliminar RET exclusivamente (instrucciones ...RET).
- Con ambas optimizaciones.
- Sin ninguna optimización.

La tabla 26 muestra la máxima ocupación de la pila de estado en cada una de estas situaciones, medida en número de posiciones ocupadas.

	sólo ..NE	sólo ..RET	ambas	ninguna	ahorro máximo
D_{max}(NORM)	8039	30	29	8052	99.6%
D_{max}(PERM)	1517	167	166	1672	99%
D_{max}(PRIMOS)	1004	403	401	1211	66.8%
D_{max}(NFIB)	25	42	24	44	45%
D_{max}(ROUND)	22	17	15	8025	99.8%

- tabla 26 - Ocupación de la pila de estado

En esta tabla se observa cómo repercuten las optimizaciones de código, de forma muy significativa, en la ocupación de esta pila. En particular, cuando el código se optimiza al máximo (se aplican las dos optimizaciones), esta ocupación baja, con respecto a la obtenida sin optimización alguna, desde un 45.5% en NFIB hasta un 99.8% en ROUND, ya que en este último, el número de optimizaciones posibles es mayor.

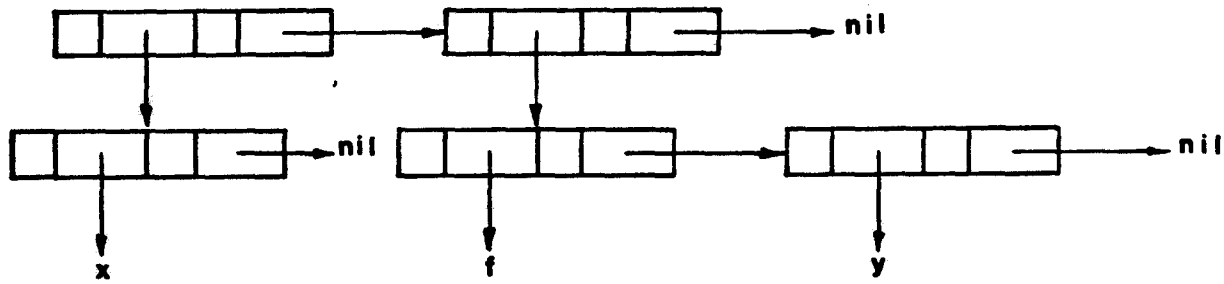
5.3.-UTILIZACION DE LA ESTRUCTURA "VECTOR" EN EL CONTEXTO

Sobre la utilización de la estructura "vector" para representar el contexto, se han realizado dos tipos de evaluación. En primer lugar, la cantidad de memoria ocupada por el contexto de evaluación, y en segundo lugar, la referente al número de accesos a memoria realizados por las instrucciones que acceden a esta información (LD n, v y sus optimizaciones).

Se ha hecho un estudio comparativo de las medidas obtenidas, utilizando contexto de listas y contexto con vectores, cuyos resultados se muestran en los apartados que siguen.

5.3.1.- OCUPACION DE MEMORIA

El estudio de la ocupación de Memoria se ha realizado en base a la ocupación máxima del "esqueleto" del contexto durante la ejecución de cada uno de los programas. El esqueleto del contexto ((x) (f y)), utilizando listas, se representa gráficamente de la forma:

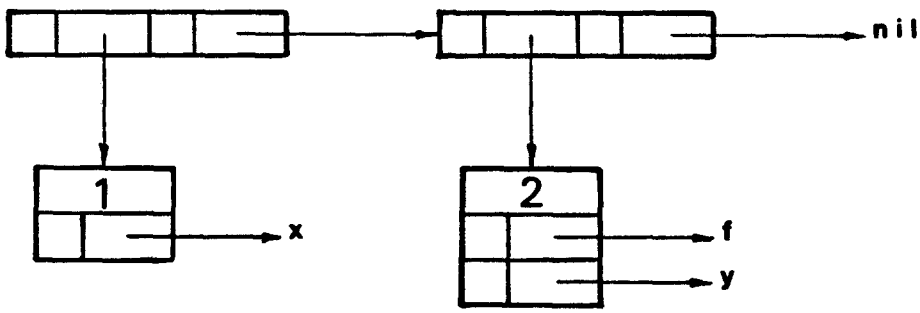


y ocupa 5 celdas de 48 bits.

Este mismo contexto, utilizando vectores, lo hemos representado de la forma

([x] [f y])

gráficamente:



y ocupa 2 celdas de 48 bits y 5 de 24 bits.

La tabla 27 muestra la ocupación máxima del esqueleto del contexto para cada uno de los programas prototipo utilizados. Los datos se dan en número de bits para ambos modelos de representación: listas y vectores. El primero utiliza celdas de 48 bits para representar, tanto la información referente al primer elemento o "car" de la lista, como al resto de ella, o su "cdr". Sin embargo, el segundo, utiliza celdas de 48 bits para formar la lista de vectores y, palabras de 24 bits para almacenar la longitud del vector y para cada uno de sus elementos, en los que vienen representados el tipo de información que contienen (8 bits) y la información o dirección correspondiente (16 bits).

	LISTAS		VECTORES				AHORRO
	celdas	bits	celdas	elementos	longitudes	bits	% BITS
NORM	14	672	3	11	3	480	28%
PERM	15	720	5	10	5	600	17%
PRIMOS	12	576	4	8	4	480	17%
NFIB	5	240	2	3	2	216	0.1%
ROUND	12	576	2	10	2	384	33%

- tabla 27 - Ocupación máxima del contexto

Vemos en esta tabla como, por ejemplo, el programa NORM utiliza un máximo de 14 celdas de 48 bits para representar el contexto en S1O, mientras que en SVO utiliza 3 celdas de 48 bits mas 14 de 24.

El ahorro conseguido al introducir este tipo de estructura varía con la estructura del programa. Podemos comprobar de este modo que en el programa PRIMOS se consigue un 17%, mientras que en ROUND se llega hasta un 33%. Esto es debido a que el primero tiene menos definiciones que el segundo.

Los porcentajes mejorarán en programas con bloques de muchas definiciones, recursivas o no, o con aplicaciones de funciones con más de un argumento, ya que éstas introducen elementos de 24 bits en el contexto compuesto por vectores, mientras que en el de listas introducen celdas de 48 bits.

Nótese como aumenta el porcentaje conseguido con NORM con respecto al de PERM, un 28% frente a un 17%, debido a que el primero tiene bloques con más definiciones que el segundo (el primer bloque SEA-REC).

5.3.2.- ACCESO A LOS VALORES DEL CONTEXTO

La utilización de esta estructura lineal, para representar el contexto, surge realmente, como ya anticipamos en la sección 3.3, para hacer más rápido el acceso a sus valores, por parte de las instrucciones LD n v, haciendo que éste acceso no sea totalmente secuencial, como ocurría en la

representación de listas.

Vemos a continuación cómo afecta esta estructura al número de accesos a memoria, que es preciso realizar, a lo largo de la ejecución de un programa, para localizar valores del contexto. Se han tenido en cuenta, el número de accesos que realizan estas instrucciones en los sistemas S10 y SV0, y que se muestran a continuación:

	LD n v	LD0 v	LD00
S10	n+v+2	v+2	2
SV0	n+2	2	2

El número de instrucciones LD ejecutadas, para cada uno de los programas utilizados, son los que se muestran en la tabla 28.

	LD n v	LD0 v
NORM	151995	8003
PERM	19178	21295
PRIMOS	26190	302
NFIB	21890	1
ROUND	13319	31588

- tabla 28 - Instrucciones LD n,v y LD0 v ejecutadas

No se recogen en esta tabla las instrucciones LD00, ya que no existe ninguna diferencia en cuanto al número de accesos que éstas deben realizar en ambos sistemas.

El número total de accesos, realizados por estas instrucciones, en ambos sistemas (S10 y SV0) son los que se muestran en la tabla 29.

	NACC_S10	NACC_SV0	% NACC
NORM	1011985	759983	25%
PERM	201217	158950	20%
PRIMOS	172930	170301	1.5%
NFIB	153237	153236	0%
ROUND	263942	170371	35%

- tabla 29 - Accesos a memoria realizados por LD n, v y LD0 v

Se observa en esta tabla que, en la ejecución de NFIB no se consigue ahorro alguno. Esto es debido a que prácticamente sólo contiene instrucciones LD n,v y "v" es cero en todos los casos. Nótese que la variación en el número de accesos de S10 a SV0 depende únicamente del valor de "v". Con el programa PRIMOS se consigue solo un ahorro del 1.5%, debido a que el número de instrucciones LD n,v es mucho mayor que el de instrucciones LD0 v y además "v" toma generalmente el valor cero.

Esta variación aumenta en los otros tres programas, consiguiéndose una mejora de hasta un 35% en ROUND debido al gran número de instrucciones LD0 v que ejecuta. La mejora conseguida en estos últimos se debe a que son programas con bloques formados por más definiciones.

Ya que existen otras instrucciones que realizan accesos a la memoria de celdas y en las que no repercute la representación del contexto (LD00, LDF, LDR, VRF, JF, J, CAR, CDR e IMP en los casos de error), cabe preguntarse, si la frecuencia dinámica de las instrucciones LD y LD0 es suficientemente alta con respecto a la de estas instrucciones, como para que este ahorro tenga una repercusión significativa en la mejora del rendimiento del sistema.

La tabla 30 muestra la frecuencia dinámica relativa de cada una de las instrucciones que realizan accesos a memoria, para cada uno de los programas.

programa	LD00	LDF	LDR	VRF	JF	J	CAR	CDR	LD0	LD
NORM	10.6	0	10	0	2.2	0	4.8	9.6	0.6	12.2
PERM	7.1	0.7	14	0	3.9	0	5.4	5.1	6.7	5.9
PRIMOS	12.7	0.1	13	0.1	4	0	8	4	0.1	8.4
NFIB	12.5	0	6.2	0	6.2	0	0	0	0	6.1
ROUND	8.4	0	13	0	3.8	0	10	4.8	9	3.3

- tabla 30 - Frecuencia dinámica de las instrucciones que acceden a memoria

Estos datos se resumen en la tabla 31, donde:

- IAM representa el porcentaje total de instrucciones que acceden a memoria
- IAC representa el porcentaje de instrucciones que acceden al contexto, exceptuando LD00
- IAC/IAM representa el porcentaje de las instrucciones que acceden a memoria, que se ven afectadas por la representación del contexto.

programa	IAM	IAC	IAC/IAM
NORM	48	12.8	26.6
PERM	48.8	12.6	25.8
PRIMOS	50.3	8.5	16.8
NFIB	31	6.1	19.7
ROUND	52.3	12.3	23.5

- tabla 31 -

Vemos en esta tabla, que las instrucciones LD n,v y LD0 v representan de un 16.8% a un 26.6% del total de las instrucciones que acceden a la memoria. Creemos que, estos porcentajes son suficientemente apreciables como para afectar de forma significativa a la mejora del rendimiento del sistema.

Resumiendo, los resultados obtenidos en este apartado muestran que la utilización de la estructura vector, para la representación del contexto de evaluación, será tanto más ventajosa cuanto mayor sea el número de definiciones en los bloques de los programas y el número de argumentos de las funciones utilizadas. Además, reduce el número de accesos a memoria debido a que utiliza el argumento "v" de las instrucciones LD n,v y LDO v, como índice para acceder de forma directa a un valor del contexto.

Estas consideraciones nos hacen pensar en la estructura "vector", como la más adecuada para representar el contexto de evaluación.

CAPITULO 6

CONCLUSIONES

6. CONCLUSIONES

Las conclusiones más importantes de este trabajo son las siguientes:

Estudiados los aspectos de representación interna de la información en los sistemas funcionales y comprobada la ineficiencia de la utilización de listas para representar ciertos tipos de información, hemos planteado y evaluado estructuras de representación lineales para el código de máquina y el contexto de evaluación de los programas funcionales, que proporciona una mejora significativa en la utilización de los recursos de la máquina.

En lo referente al código, hemos realizado un modelo de máquina virtual SL0, que utiliza una estructura lineal para representar las instrucciones de máquina, para el que hemos definido un juego completo de instrucciones y en el que se han incluido una serie de optimizaciones que mejoran sustancialmente la carga del sistema. Una vez evaluado este modelo y comparado con un modelo similar cuya única estructura de representación son las listas, se ha observado que:

- El modelo desarrollado mejora sensiblemente el tiempo de ejecución de los programas, ya que utiliza un registro para acceder a la siguiente instrucción a ejecutar, en lugar de hacerlo siguiendo apuntadores.

- El código del programa ocupa un espacio considerablemente menor, debido a que sólo se representa lo necesario, es decir, las instrucciones, en contra de lo que ocurre en el modelo de listas donde además hay que representar el modo de acceder a ellas (apuntadores).

- El hecho de no poder recolectar el código, no influye de forma negativa en el sistema, ya que hemos comprobado que la fracción de memoria que se podría recolectar, es realmente insignificante frente al total de las celdas de memoria accesibles. Por otra parte, comprobado que el tiempo de recolección es proporcional al número de celdas accesibles, la no recolección del código hace que este tiempo sea menor y, por lo

tanto, tambien lo es el tiempo total de ejecución.

En lo referente a la representación del contexto de evaluación, hemos introducido un nuevo tipo de objeto "vector", obteniendo un nuevo modelo de máquina virtual SVO, cuyas ventajas más importantes son:

- Una menor ocupación de memoria que con el uso exclusivo de listas, ya que los valores del contexto están agrupados en posiciones contiguas de memoria (vectores) y no es necesario el uso de apuntadores.

- Un acceso más rápido a los valores del contexto, ya que para localizarlos basta con utilizar un índice para realizar un acceso directo.

Hemos desarrollado un sistema funcional concurrente, basado en la especificación de múltiples ficheros de salida, que permite aprovechar el paralelismo inherente a los lenguajes funcionales de modo totalmente transsparente al usuario. Este sistema está compuesto por:

- Un modelo de evaluación, transparente a la forma en que se gestiona la memoria del sistema, en el que las distintas tareas del sistema cooperan en la "reducción" del programa, con un mecanismo sencillo y potente de sincronización y comunicación entre tareas que permite detectar y resolver las dependencias existentes entre ellas.

- Una forma de responder a los errores en tiempo de ejecución, que permite continuar normalmente a las tareas no involucradas y hacer que las que lo están acaben su ejecución del mismo modo que si no hubiese existido error alguno, siendo la única diferencia el resultado obtenido.

- Un modelo de gestión de memoria, transparente al algoritmo de recolección y al mecanismo de evaluación utilizados, y basado en la gestión centralizada de una memoria común que se concede a las tareas según lo necesitan mediante el paso de mensajes, que permite que las tareas no tengan que competir por el espacio libre que poseen.

Tanto los modelos de máquina virtual SLO, SVO como el sistema

funcional de evaluación multitarea han sido simulados mediante programas escritos en lenguaje C en el VAX-11/750 de la Facultad de Informática de Madrid.

Como conclusión global podemos destacar el satisfactorio comportamiento del sistema funcional obtenido y la mejora obtenida con respecto a sistemas ya existentes, que creemos supone una aportación importante al desarrollo e investigación de sistemas programados mediante lenguajes funcionales.

BIBLIOGRAFIA

BIBLIOGRAFIA

[1] Backus, J.
Can Programming be liberated from the Von Neumann style?. A Functional Style and its Algebra of Programs.
Comm. ACM, 21(8), aug. 1978, 613-641

[2] Baker, H. y Hewit, C.
The incremental garbage collection of processes.
SIGPLAN Notices, vol. 12, aug. 1977, pp. 55-59

[3] Baker, H.
List processing in Real Time on a Serial Computer
AI working Paper 139, MIT AI Lab. Feb. 1978

[4] Ben-Ari, M.
Algorithms for On-the-fly Garbage collection.
ACM Trans. on Programming Languages and Systems, vol. 6, no. 3, july 1984, 333-344.

[5] Bobrow, D.G. y Clark, D.W.
Compact Encodings of list Structure.
ACM Trans. on Programming Languages and Systems, vol. 1, no. 2, oct. 1979, 266-286

[6] Burge, W.H.
Recursive Programming Techniques.
Addison-Wesley, 1975

[7] Burton, F.W. y Sleep, M.R.
Executing Functional Programs on a Virtual Tree of Processors.
Proc. Functional Programming Lang. and Comp. Arch. , oct. 1981

[8] Clark, D.W. y Green C.C.
An empirical Study of List structure in Lisp
Comm. of ACM, Feb. 1977, vol. 20, no. 2, pp-78-87

[9] Clark, D.W.
Measurements of dinamic list structure use in Lisp.
IEEE Trans. on Software Eng., vol. SE-5, no.1, jan. 1979, 51-59

[10] Cheney, C.J.
A nonrecursive list compacting algorithm
Comm. of ACM 13, 11, Nov. 1970, 677-678

[11] Cohen, J.
Garbage collection of linked data structures.
Computing Surveys, vol 13, no. 3, sept, 1981.

[12] Cohen, J., Nicolau, A.
Comparison of Compacting Algorithms for Garbage Collection.
ACM Trans. on Programming Languages and Systems, vol. 5, no. 4,
oct. 1983, 532-553.

[13] Dawson, J.L.
Improved effectiveness from a real time garbage collector.
ACM Symp. on Lisp and Functional Programming, 1982, pp. 159-167

[14] Dijkstra, E.W. y Lamport L.
On-the-fly garbage collection: An exercise in Cooperation.
Comm. ACM, vol. 21, no. 11, nov. 1978, 966-975

[15] Fenichel, R.R. y Yochelson.
J.C. A LISP garbage-collector for virtual-memory computer systems.
Comm. ACM 12,11 (Nov 1969) 611-612

[16] Friedman, D.P. y Wise, D.S.
The impact of Applicative Programming on Multiprocessing.
Proc. 1976 Int. Conf. on Parallel Processing. 263-272

[17] Friedman, D.P. y Wise, D.S.
CONS should not Evaluate its arguments
Autómatas, Languages & Programming. Edinburg Univ. Press, 1976, pp. 257-284

[18] Friedman, D.P. y Wise, D.S.
An environment for multiple-valued recursive procedures
Proc. 2nd Colloque sur la Programmation, Springer-Verlag, 1976

[19] Friedman, D.P. y Wise, D.S.
Aspects of Applicative Programming for File Systems.
ACM Sigplan Notices, mar. 1977, 12(3), 41-55

[20] Friedman, D.P. y Wise, D.S.
Aspects of Applicative Programming for Parallel Processing.
IEEE Trans. on Comp., 27(4), apr. 1978, 289-296

[21] Friedman, D.P. y Wise, D.S.
Functional Combination
Computer Languages, vol.3 pp. 31, Pergamon Press, 1978

[22] Grit, D. y Page, R.
Eager Evaluation of functional Programs and a Supporting Interconection Structure.
Dep. of Comp. Science, Colorado State, FT. Collins, CO. 1981

[23] Grit, D. y Page, R.
Deleting Irrelevant Tasks in a Expression Oriented Multiprocessor System.
ACM Trans. on Programming Languages and Systems, vol. 3, no. 1, Jan. 1981, pp. 49-59.

[24] Hansen, W.J.
Compact List Representation: Definition, Garbage collection and System implementation.
Comm. ACM, vol. 12, no.9, sept. 1969

[25] Hart, T.P. y Evans, T.G.
Notes on implementing Lisp for M-460 computer.
Ref. [1], 191-203, 1966

[26] Henderson, P.
A lazy Evaluator
Proc. 3rd. ACM Symp. on Principles of Programming Languages.
Atlanta, Ga., jan. 1976, 95-103

[27] Henderson, P.
Functional Programming. Application & Implementation.
Prentice Hall International, Englewood Cliffs, N.J. 1980

[28] Henderson, P. Jones G.A y Jones S.B.
The LispKit Manual
Oxford Univ. Computing Lab. vol 2 Tech. Monograph PRG-32(2)

[29] Hudak, P., Keller, R.M.
Garbage Collection and Task Deletion in Distributed applicative processing systems.
ACM Symp on Lisp and Functional Programming, 1982, pp. 168-178.

[30] Keller, Lindstrom, Patil.
An Architecture for a loosely-coupled parallel processor.
University of Utah, Dept. of Computer Science. Tech. rept.
UUCS-78-105 (1978).

[31] Keller, R.M.
Divide and CONCer: Data Structuring in Applicative Multiprocessing Systems.
Proc. Lisp Conf. Aug. 1980, pp. 196-202

[32] Knuth, D.E.
The Art of Computer Programmming, Vol I: Fundamentals Algorithms.
Addison-Wesley, Reading, Mass., 1968

[33] Lamport, L.
Garbage collection with multiple Processes: An exercise in Parale-
lism
Massachussetts Computer Associates, CA-7602-2511, 1976

[34] Landin, P.J.
The Mechanical Evaluation of Expressions.
Computer Journal, apr. 1964, 6(4), 308-320

[35] Mañas, J.A.
Arquitecturas de Computadores de Reducción
Tesis Doctoral, Fac. Informática de Madrid, Jun. 1983

[36] Mañas, J.A. y Lafuente, J.A.
Elección de una Red de Procesadores para reducir lenguajes
funcionales.
Inf. Interno 6.1. Fac. de Informática de S. Sebastián, Marzo 1984

[37] Mañas, J.A. y García, M.I.
L10: Un lenguaje puramente funcional.
Inf. Interno Fac. Informática de Madrid, Dic. 1984

[38] McCarthy, J.
Recursive functions of symbolic expressions and their computation
by machine.
Comm. ACM 3, 4 (Apr. 1960) 184- 195

[39] McCarthy, J. y otros
Lisp 1.5 Programmer's Manual.
MIT Press, Cambridge Mass, 1965

[40] Newel, A.
Information Processing Language V Manual.
Prentice Hall, Englewood Cliffs, N.J., 1961.

[41] Steele, G.L.
Multiprocessing Compactifying garbage collection
Comm. ACM sept. 1975, vol 18, no. 9, 495-508

[42] Steele, G.L. y Sussman, G.J.
Design of a Lisp based Microprocessor
Comm. ACM, vol. 23, no. 11, Nov. 1980

[43] Sussman, G.J. y otros
Scheme 79 - Lisp on a Chip
Computer, jul. 1981, 14(7), pp. 10-21

[44] Treleaven, P.C.
Data Driven & Demand Driven Computer Architecture.
Univ. of Newcastle upon Tyne, Computing Lab. Tech. Rep. Series,
no. 168, jun. 1981

[45] Turner, D.A.
An Implementation of SASL
Univ. of St. Andrews, Dep. of Comp. Sci. Report TR/75/4.

[46] Turner, D.A.
A New Implementation Technique for Applicative Systems
Software Practice and Experience, vol. 9, 31-39, 1979

[47] Van der Poel, W.L.
A Manual of HISP for the PDP-9.
Technical U. Delft, Netherlands.

[48] Weinreb, D y Moon, D.
Lisp Machine Manual.
Symbolics Inc. July 1981

[49] Weizenbaum, J.
Knotted list structures.
Comm. ACM, 5, 3, March, 1962, 161-165.

A1. JUEGO DE INSTRUCCIONES DE MAQUINA

1.- ECONS

x,s, e pc d --------> s (x.e) pc+1 d

2.- NCONS

s e pc d --------> s (NIL.e) pc+1 d

3.- ERPL

x,s (NIL.e) pc d --------> s (x.e) pc+1 d

4.- ECDR

s (x.e) pc d --------> s e pc+1 d

5.- LD n v

s e pc d -------->

sea x= v+1 (n+1 (e))

cond

si x= 1(rz(e1, pc1).b) ent

--------> s e1 pc1 E:e, A:(rze.b), R:pc+3, d

si x= rze ent

--------> "*dci*",s e pc+3 d

sino

--------> x,s e pc+3 d

6.- LDRET n v

s e pc d -------->

sea x= v+1 (n+1 (e))

cond

si x= 1(rz(e1, pc1).b) ent

--------> s e1 pc1 E:e, A:(rze.b), d

si x= rze ent

--------> "*dci*",s e RET d

sino

--------> x,s e RET d

7.- LDNE n v

Igual a LD n v sin guardar contexto, E:e

8.- LDO v

Igual a LD n v cambiando pc+3 por pc+2

9.- LD0RET v

Igual a LDRET n v cambiando pc+3 por pc+2

10.- LD0NE v

Igual a LDNE n v cambiando pc+3 por pc+2

11.- LD00

Igual a LD n v cambiando pc+3 por pc+1

12.- LD00RET

Igual a LDRET n v cambiando pc+3 por pc+1

13.- LD00NE

Igual a LDNE n v cambiando pc+3 por pc+1

14.- LDCN n

s e pc d --------> n,s e pc+2 d

15.- LDZ

s e pc d --------> 0,s e pc+1 d

16.- LDCC c

s e pc d --------> c,s e pc+2 d

17.- LDCB bool

s e pc d --------> bool,s e pc+2 d

18.- LDN

s e pc d --------> NIL,s e pc+1 d

19.- LDF pc1

s e pc d --------> apt(e, pc1),s e pc+2 d

20.- LDR pc1

s e pc d --------> rz(e, pc1),s, e pc+2 d

21.- AP

x,v,s e pc d -------->

cond

si x= apt(e1,pc1) ent

--------> s (v.e1) pc1+1 E:e, R:pc+1, d

si número(x) ent

cond

si x= 0 ent ------> 1(v),s e pc+1 d

si x= -1 ent -----> 1(v),s e RCD R:pc+1, d

si x= 1 ent ------> 1(v),s e RCA R:pc+1, d

sino ---------> 1(v),x,s e RCN R:pc+1, d

sino

--------> "x *nea*,s e pc+1 d

22.- APRET

```
x,v,s   e   pc   d   -------->
cond
si x= apt(el,pcl) ent
        -------->   s   (v.el)   pcl+1   E:e, d
si número(x) ent
        cond
        si x= 0 ent ------>   l(v),s   e   RET   d
        si x= -l ent ----->   l(v),s   e   RCD   d
        si x= l ent ------>   l(v),s   e   RCA   d
        sino        --------->   l(v),x,s   e   RCN   d
sino
        -------->   "x *nea*",s   e   RET   d
```

23.- APNE

Igual a AP sin guardar contexto, E:e

24.- VRF n

```
x,v,s   e   pc   d   -------->
cond
si x= apt(el,pcl) ent
        sea n'= contenido (pcl)
        cond
        si n < n' ent
                  -------->   "$A *iar*",v,s   e   pc+2   d
        si n > n' ent
                  -------->   "$A *ear*",v,s   e   pc+2   d
        sino   -------->   x,v,s   e   pcl+2   d
si número(x) ent
        si n ◇ l ent
          cond
          si x= 0 ent ------>   "0 *ear*",v,s   e   pc+2   d
          si x= -l ent ----->   "-l *ear*",v,s   e   pc+2   d
          si x= l ent ------>   " l *ear*",v,s   e   pc+2   d
          sino   --------->   "x *ear*",v,s   e   pc+2   d
        sino
                -------->   x,v,s   e   pc+2   d
sino
        -------->   x,v,s   e   pc+2   d
```

25.- RET

```
        s   e   pc   d   -------->
        si pila D vacía ent
                        -------->   s   e   pc   vacía
        sino
                        -------->   s   NIL   pc   d   -------->
                loop
                   sea d= d1,d2 y s= x,s1
                   cond
                        si d1= R:pc1 ent
                                -------->   s   e   pc1   d2
                                        exit
                        si d1= A:(rze.b) ent
                                -------->   s   e   pc   d2
                                        y (rze.b) ---> (x.b)
                        si d1= D:(b.rze) ent
                                -------->   s   e   pc   d2
                                        y (b.rze) ---> (b.x)
                        sino    -- d1= E:e1
                                ------->   x,s   e1   pc   d2
                end-loop
```

26.- JF pc1

```
        x,s   e   pc   d   -------->
        cond
        si x= "Falso" ent
                        -------->   s   e   pc1   d
        si x= "Cierto" ent
                        -------->   s   e   pc+2   d
        sino
                        -------->   "(si x)", s   e   pc1-2   d
```

27.- J pc1

```
        s   e   pc   d   -------->   s   e   pc1   d
```

28.- CAR

```
        x,s   e   pc   d
        cond
        si x no es una lista de tipo CONS ent
                        -------> "1 x",s   e   pc+1   d
```

si x= (rz(el,pcl).b) ent

--------> s el pcl E:e, A:(rze.b),R:pc+1,d

si x= (rze.b) ent

--------> "*dci*",s e pc+1 d

sino

--------> 1(x),s e pc+1 d

29.- CARRET

x,s e pc d

cond

si x no es una lista de tipo CONS ent

--------> "1 x",s e RET d

si x= (rz(el,pcl).b) ent

--------> s el pcl E:e, A:(rze.b), d

si x= (rze.b) ent

--------> "*dci*",s e RET d

sino

--------> 1(x),s e RET d

30.- CARNE

Igual que CAR sin guardar contexto, E:e

31.- CDR

x,s e pc d

cond

si x no es una lista de tipo CONS ent

--------> "-1 x",s e pc+1 d

si x= (a.rz(el,pcl)) ent

--------> s el pcl E:e, D:(a.rze),R:pc+1,d

si x= (a.rze) ent

--------> "*dci*",s e pc+1 d

si x= (a.pf()) ent

si fin de fichero ent

--------> NIL, s e pc+1 d

sino

sea b= primer elemento del fichero

--------> (b.pf'()),s e pc+1 d

y (a.pf()) ----> (a b.pf'())

sino

--------> -1(x),s e pc+1 d

32.- CDRRET

x,s e pc d

cond

si x no es una lista de tipo CONS ent

--------> "-1 x",s e RET d

si x= (a.rz(e1,pc1)) ent

--------> s e1 pc1 E:e, D:(a.rze), d

si x= (a.rze) ent

--------> "(*dci*)",s e RET d

si x= (a.pf()) ent

si fin de fichero ent

--------> NIL, s e RET d

sino

sea b= primer elemento del fichero

--------> (b.pf'()),s e RET d

y (a.pf()) ----> (a b.pf'())

sino

--------> -1(x),s e RET d

33.- CDRNE

Igual a CDR sin guardar contexto, E:e

34.- CNR

v,i,s e pc d --------->

cond

si i > 1 ent

--------> v, i-1, s e RDN d

si i= 1 ent

--------> v,s e RCA d

si i= -1 ent

--------> v,s e RCD d

si i < -1 ent

--------> v, i+1,s e RDN d

35.- CONS

a,b,s e pc d --------> (a.b),s e pc+1 d

36.- FICH

id,s e pc d -------->

si existe el fichero de nombre "id" y es posible abrirlo ent

sea "a" el primer caracter del fichero

--------> (a.pf()),s e pc+1 d

sino

--------> "(FICH id)",s e pc+1 d

37.- IMP

x,s e pc d -------->

cond

si x= lista de tipo CONS ent

--------> x,x,s e RIC R:pc+1,d

si x= lista de tipo ERROR ent

sea x= (x1.x2)

--------> x1,x2,s e RIE R:pc+1,d

sino

--------> s e pc+1 d

y si x= número ent lo imprime

si x= caracter ent lo imprime

si x= booleano ent lo imprime

si x= apunte ent imprime $A

si x= rz ent imprime $R

si x= rze ent imprime $E

si x= pf ent imprime $P

38.- ATM

x,s e pc d ------> bool,s e pc+1 d (1)

39.- NUL

x,s e pc d ------> bool,s e pc+1 d (1)

40.- ADD

b,a,s e pc d ------> a+b,s e pc+1 d (2)

41.- SUB

b,a,s e pc d ------> a-b,s e pc+1 d (2)

42.- MUL

b,a,s e pc d ------> a*b,s e pc+1 d (2)

43.- DIV

b,a,s e pc d ------> a/b,s e pc+1 d (3)

44.- MOD

b,a,s e pc d ------> a mod b,s e pc+1 d (3)

45.- NEG

a,s e pc d ------> -a,s e pc+1 d (4)

46.- AND

b,a,s e pc d ------> a and b,s e pc+1 d (5)

47.- OR

b,a,s e pc d ------> a or b,s e pc+1 d (5)

48.- XOR

b,a,s e pc d ------> a xor b,s e pc+1 d (5)

49.- NOT

a,s e pc d ------> not a,s e pc+1 d (6)

50.- EQ

b,a,s e pc d ------> a= b,s e pc+1 d (7)

51.- NE

b,a,s e pc d ------> a<>b,s e pc+1 d (7)

52.- LT

b,a,s e pc d ------> a< b,s e pc+1 d (7)

53.- LE

b,a,s e pc d ------> a<=b,s e pc+1 d (7)

54.- GT

b,a,s e pc d ------> a> b,s e pc+1 d (7)

55.- GE

b,a,s e pc d ------> a>=b,s e pc+1 d (7)

(1) si x es una lista de tipo ERROR ent
sea 'op' la operación correspondiente
--------> "(op x)",s e pc+1 d

(2) si a y/o b no son átomos numéricos ent
sea 'op' la operación correspondiente
--------> "(op a b)",s e pc+1 d

(3) si a y/o b no son átomos numéricos o si b= 0 ent
sea 'op' la operación correspondiente
--------> "(op a b)",s e pc+1 d

(4) si a no es un átomo numérico ent
sea 'op' la operación correspondiente
--------> "(op a)",s e pc+1 d

(5) si a y/o b no son los átomos booleanos C y F ent
sea 'op' la operación correspondiente
--------> "(op a b)",s e pc+1 d

(6) si a no es el átomo booleano C o F ent
sea 'op' la operación correspondiente
--------> "(op a)",s e pc+1 d

(7) si a y/o b son listas de tipo ERROR ent
sea 'op' la operación correspondiente
--------> "(op a b)",s pc+1 d

Las instrucciones incluidas en las rutinas del sistema son las que se indican a continuación:

```
RCD: CDRRET

RCA: CARRET.

RDN: CDRNE
RCN: CNR

RIC: CARNE              -- Impresión de una lista de tipo "cons"
     IMPNE
     CDRNE
     IMPRET

RIE: IMPNE              -- Impresión de una lista de tipo "error"
     IMPRET

RII: IMP                -- Código inicial de la pila D
     RET
```

APENDICE 2

PROGRAMAS PROTOTIPO

1.- NORM: Obtención de números aleatorios distribuidos normalmente

```
sea-rec (sel ((1 no), normal))  -- resultado

  normal = red6 (unif)          -- definiciones

  unif   = sea-rec (s : ((map (sig)) unif))
             sig = función (x)
                   mod( ((a * x) + b), (2 * c) ) - c
             a   = 101
             b   = 97
             c   = 100
             s   = 13

  map    = función (f)
           sea-rec g
             g = función (x)
                 si x = NIL  ent NIL
                 si no f (1 (x)) : g (-1 (x))

  red6   = función (n1, n2, n3, n4, n5, n6.n)
           (n1 + n2 + n3 + n4 + n5 + n6) : red6 (n)

  sel    = función (n, m)
           si n = 0 ent "cr"
           si no 1 (m) : " " : sel (n-1, -1 (m))

  no     = fich (tt:)
```

2.- PERM: Permutaciones de los átomos de la lista "lis"

```
sea-rec perm (lis)                          -- resultado

  perm = función (x, n)                     -- definiciones
         sea-rec si -1 (n) = NIL ent 1 (n) : "cr" : NIL
                 si no p (n)
           p = función (x.s)
               sea si s = NIL ent u
                   si no append (u, p (s))
               u = q (x, quita (x, n))
           q = función (x1, n1)
               consl (x1, perm (n1))

  consl = función (x, n)
          si n = NIL ent  (x : 1 (n)) : consl (x, -1 (n))

  quita = función (x, n)
          si x = 1 (n) ent -1 (n)
          si no 1 (n) : quita (x, -1 (n))

  append = función (11, 12)
           si 11 = NIL ent 12
           si no 1 (11) : append (-1 (11), 12)

  lis    = fich("numeros.dat")
```

3.- PRIMOS: Cálculo de la lista de números primos.

```
sea-rec sel ( (1 no), (p (n, 2)) )        -- resultado

  p = función (1)                         -- definiciones
        (1 1) : p (s (1 1)) : (-1 1)

  s = función (p)
        sea-rec sp
            sp = función (1)
                 si mod ( (1 1), p) = 0 ent sp(-1 1)
                 si no (1 1) : sp (-1 1)

  n = función (x)
        x : n (x + 1)

  sel = función (n, m)
          si n = 0 ent "cr"
          si no (1 m) : " " : sel ( (n - 1), (-1 n))

  no = fich (tt:)
```

4.- NFIB: Número de aplicaciones para encontrar el elemento n-ésimo de la serie de Fibonacci.

```
sea-rec nfib (1 no)                       -- resultado

  nfib = función (n)    -- definiciones
           si n <= 1 ent 1
           si no nfib (n - 1) + nfib (n - 2) + 1

  no = fich (tt:)
```

5.- ROUND: Cálculo de la lista de los números cuyos únicos factores primos son 2, 3 6 5

```
sea-rec sel ((1 no), x)

   x = 1 : merge2 (x2, merge2 (x3, x5))

   x2 = por (2, x)

   x3 = por (3, x)

   x5 = por (5, x)

   merge2 = función (a, b)
            cond
            si (1 a) = (1 b) ent merge2 (a, (-1 b))
            si (1 a) < (1 b) ent (1 a) : merge2 ( (-1 a), b) )
            si no (1 b) : merge2 (a, (-1 b) )

   por = función (n, x)
            (n * (1 x)) : por (n, (-1 x))

   sel = función (n, m)
          si n = 0 ent "cr"
          si no (1 m) : " " : sel ( (n - 1), (-1 n))

   no = fich (tt:)
```